Lezione 1 - Introduzione

Calcolabilità: capire quali problemi possono essere risolti *automaticamente*, cosa significa *risolvere un problema* e cosa *è un problema*
Complessità: capire quali dei problemi che possono essere risolti possono essere realmente risolti.

Problemi

Def.
Un **problema** è la descrizione di un insieme di *dati*, collegati da una serie di relazioni, associata alla richiesta di derivare da essi un altro insieme di parametri, che chiamiamo *soluzione*.

Un'istanza di un problema è un particolare insieme di valori associati ai dati.

> **Es.**
>
> - "Quanto fa 5+2?" è un istanza del PROBLEMA SOMMA, ossia, dati due numeri n e k, calcolare $s = n + k$.
> - "Quanto misura l'area di un rettangolo con $b = 12$ e $h = 4$?" è un istanza del PROBLEMA AREA, ossia, dato un rettangolo con base b e altezza h, calcolare l'area A del rettangolo.

Trovare la soluzione di un'istanza

A volte è semplice trovare la soluzione di un'istanza di un problema, ad esempio per il **PROBLEMA SOMMA**, l'istanza con $n = 2$ e $k = 5$ è semplice. Altre volte è più complicato, pensiamo al problema somma con $n = 524332952$ e $k = 78435020353$.
A volte è proprio **impossibile**, ad esempio trovare un $n \in \mathbb{R}$ tale che $n = \sqrt{-4}$.

Quando l'istanza di un problema non ha soluzione diciamo che è un'**istanza negativa**

Risolvere un problema

Risolvere un problema significa individuare un *metodo* che sappia trovare la **soluzione di ogni istanza positiva del problema** e che sappia riconoscere un'*istanza negativa*, ovvero trovare un procedimento che, data qualunque istanza, indichi la sequenza di azioni che devono essere svolte per trovare la soluzione dell'istanza, o concludere che l'istanza non ha soluzione.

Bisogna ora rispondere a qualche domanda:

- cos'è un procedimento?
- cos'è un'azione?
- chi deve eseguire le azioni indicate?

Def.
Un **procedimento** è la descrizione di un insieme di azioni, unita alla specifica dell'ordine con il quale queste azioni devono essere eseguite.
Le **azioni** indicate, a cui viene dato il nome di *istruzioni*, devono essere semplici (elementari), ovvero eseguibili con facilità.

> **Es.**
> Data una funzione $f : \mathbb{R} \to \mathbb{R}^+$ e dati due numeri $a, b \in \mathbb{R}$, calcolare la misura dell'area della regione di piano compresa fra la funzione, l'asse x e l'asse y e la retta $y = a$ e $y = b$.
>
> PROCEDIMENTO:
>
> 1. calcola la funzione primitiva $F(x)$ di $f(x)$
> 2. calcola $F(b) - F(a)$

Quello indicato nell'esempio è un procedimento che risolve il problema, ma l'istruzione "calcola la funzione primitiva $F(x)$ di $f(x)$" è davvero un'**istruzione elementare?**
Dipende da chi *deve eseguire le azioni indicate*, nel caso di un matematico l'istruzione è elementare, mentre per un bambino delle elementari ovviamente no.

Istruzioni elementari

Turing osservò che, indipendentemente dall'esecutore, un'istruzione per poter essere definita *elementare*, deve avere le seguenti caratteristiche:

- deve essere scelta in un insieme di "poche" istruzioni
- scegliere l'azione da un insieme di "poche" azioni
- deve essere poter eseguita ricordando una quantità limitata di dati
  Le caratteristiche individuate, indicano come istruzione elementare un'operazione che possa essere eseguita a mente.

Esempio con il problema somma

> Dati due interi n e k, calcolare il numero $n + k$.

Vogliamo quindi progettare un procedimento che risolva questo problema. Siccome calcolare la somma di due interi è facile, potremmo pensare che l'istruzione "calcola $n+k$" sia un'*istruzione elementare*, ma se assegnassimo $n=37895$ e $k=441238$, a nessuno viene in mente il risultato, questo perché la nostra memoria è limitata.
Per le somme tra i numeri da 0 a 9 abbiamo a suo tempo imparato una tabella simile

"FI/MOD II/img/img0.png" non trovato.

quindi basterebbe una tabella sufficientemente grande per risolvere l'istanza sopra, ad esempio la somma dei numeri tra 0 e 100000.

"FI/MOD II/img/img1.png" non trovato.

Questo sembrerebbe corretto per risolvere l'istanza data, ma per risolvere il problema somma, occorre indicare un *procedimento* che sappia addizionare **qualunque coppia di numeri naturali**, e questo comporterebbe quindi la costruzione di una tabella infinita, per questo motivo l'istruzione "calcola $n+k$" non può essere definita *elementare*.

Per risolvere un problema utilizziamo un procedimento che utilizza un numero limitato di operazioni elementari (somme di coppie di numeri di una sola cifra) e in cui ogni operazione elementare utilizza una quantità limitata di dati (due cifre e l'eventuale riporto).
Pensiamo quindi alla somma dei numeri in colonna, potremmo descrivere il procedimento per calcolare il risultato nel modo seguente:

1. posizionati sulla colonna più a destra e poni $r=0$
2. fintanto che leggi una coppia di cifre, esegui la somma della coppia di cifre sulle quali sei posizionato, aggiungi r a tale valore e scrivi una cifra del risultato calcolando il nuovo valore di r, e poi spostati a sinistra
   - se $r=0$ e le due cifre sono $(0,0)$, scrivi 0, poni $r=0$ e spostati a sinistra
   - se $r=1$ e le due cifre sono $(0,0)$, scrivi 1, poni $r=0$ e spostati a sinistra
   - $\vdots$
   - se $r=0$ e le due cifre sono $(9,9)$, scrivi 8, poni $r=1$ e spostati a sinistra
   - se $r=1$ e le due cifre sono $(9,9)$, scrivi 9, poni $r=1$ e spostati a sinistra
3. fino a quando leggi una sola cifra, aggiungi r ad essa e scrivi una cifra del risultato calcolando anche il nuovo valore di r e spostati a sinistra
   - se $r=0$ e l'unica cifra è 0, scrivi 0, poni $r=0$ e spostati a sinistra
   - se $r=1$ e l'unica cifra è 0, scrivi 1, poni $r=0$ e spostati a sinistra
   - $\vdots$
   - se $r=1$ e l'unica cifra è 8, scrivi 9, poni $r=0$ e spostati a sinistra
   - se $r=1$ e l'unica cifra è 9, scrivi 0, poni $r=1$ e spostati a sinistra

4. se le cifre di entrambi i numeri sono terminate, allora calcola l'eventuale ultima cifra del risultato e termina
    - se $r = 0$ e le cifre di entrambi i numeri sono terminate, allora termina
    - se $r = 1$ e le cifre di entrambi i numeri sono terminate, allora scrivi 1 e termina

Oss.
Il procedimento per calcolare la somma in colonna, non è nient'altro che una sequenza di "se sono vere **queste condizioni**, allora esegui *queste azioni*", quindi ad ogni coppia (**certe condizioni**, *queste azioni*) corrisponde un'istruzione, dove le condizioni sono ciò che viene letto e le azioni sono ciò che viene scritto.

Ora queste istruzioni sono veramente *elementari*, infatti ogni persona che sappia leggere e scrivere potrebbe eseguire il procedimento.

Le istruzioni individuate sono, però, *elementari* nel senso indicato da Turing? Ricordiamo che devono avere le seguenti caratteristiche:

- scelte in un insieme di "poche" istruzioni disponibili
- **deve scegliere l'azione da eseguire all'interno di un insieme di "poche" azioni disponibili**
- deve poter essere eseguita ricordando una quantità limitata di memoria
  Nel procedimento visto sopra le azioni eseguite sono due: scrittura della cifra e spostamento, che sono effettivamente "poche" azioni, ma è vero che il procedimento che esegue la somma ha un insieme di "poche" istruzioni disponibili, ciascuna delle quali utilizza una quantità limitata di memoria?

Spieghiamo cosa si intende con poche e quantità limitata.
Il numero di istruzioni disponibili è pari al numero di coppie di cifre moltiplicato per il numero di possibili valori per il riporto, cioè $10 \times 10 \times 2 = 200$. Per sapere quale istruzione esguire abbiamo bisogno di conoscere le cifre da sommare e il valore del riporto, ossia 3 numeri di una cifra.

Ricapitolando: per sommare qualunque coppia di interi *grande a piacere* abbiamo a disposizione 222 istruzioni fra le quali scegliere quella da eseguire utilizzando una memoria di 3 cifre. Indipendentemente da quanto sono grandi i numeri, sempre 222 istruzioni che utilizzano una memoria di 3 cifre vengono eseguite.

> **Importante**
>
> Il numero di istruzioni, azioni e la quantità di memoria necessaria sono **costanti**, ossia non dipendono dall'input

Eseguendo il procedimento, non è necessario sapere cosa significa sommare due numeri naturali, poiché le istruzioni dicono **per ogni condizione possibile, esegui queste azioni**, questo significa che l'insieme delle istruzioni non è *ambiguo*: non contiene due o più istruzioni che a partire dalle stesse **condizioni**, indica diverse *azioni*. In ogni istante *deve essere* eseguita **l'unica istruzione** che è possibile eseguire, finché non si incontra l'istruzione che dice di terminare. Quest'idea è alla base di molti linguaggi di programmazione che vengono detti *imperativi* (C, Java, Python ecc.).

Risolvere automaticamente un problema

Informalmente, risolvere automaticamente un problema, significa progettare un **procedimento** che risolve **tutte** le istanze del problema e che *può essere eseguito da un automa*, ovvero un esecutore che può non avere alcuna idea del problema né del significato delle istruzioni contenute nel procedimento.

Un nuovo linguaggio

Ripensiamo alla somma dei numeri naturali:

1. il procedimento visto è costituito di sole istruzioni "condizione -> azione"
2. in ciascuna istruzione le azioni da eseguire sono 3 (scrittura della cifra, modifica del riporto, movimento a sinistra)
3. le condizioni di ognuna delle istruzioni dipendono da due tipi di parametri, ovvero il valore del riporto e le due cifre da sommare
   Mentre le due cifre da sommare le troviamo scritte sul foglio, il valore del riporto lo teniamo a mente ad ogni coppia di cifre sommate, ovvero il nostro **stato interiore**.

In seguito a queste osservazioni possiamo scrivere il procedimento in forma più compatta, l'istruzione

- se $r = 0$ e le due cifre sono $(4, 6)$, scrivi 0, poni $r = 1$ e spostati a sinistra
  diventa

  $$\langle q_0,\ (4,6),\ 0,\ q_1,\ \mathrm{sx} \rangle$$

  dove q_0 e q_1 sono due simboli che indicano $r = 0$ e $r = 1$

L'istruzione

- se $r = 1$ e l'unia cifra è 5, scrivi 6, poni $r = 0$ e spostati a sinistra
  diventa

$$\langle q_1,\ (5,\square),\ 6,\ q_0,\ \mathrm{sx}\rangle$$
$$\langle q_1,\ (\square,5),\ 6,\ q_0,\ \mathrm{sx}\rangle$$

dove $\square$ indica che non viene letto nulla.

Le istruzioni di terminazione

- se $r = 0$ e le cifre di entrambi i numeri sono terminate, allora termina
- se $r = 1$ e le cifre di entrambi i numeri sono terminate, allora scrivi 1 e termina

  diventano

  $$\begin{matrix}\langle q_0,\ (\square,\square),\square,\ q_f,\ \mathrm{fermo}\rangle \\ \langle q_1,\ (\square,\square),1,\ q_f,\ \mathrm{fermo}\rangle\end{matrix} \$\$dovecon\$q_f\$èlo'' * statointeriore *''chepermetteall'esecutore$$

"FI/MOD II/img/img2.png" non trovato.

Quella in figura è *quasi* una descrizione informale di macchina di Turing, perché abbiamo usato tre nastri e in una macchina di Turing occorre descrivere cosa viene letto (nelle **condizioni**) e cosa viene scritto (nelle *azioni*) su **ogni** nastro, così che l'istruzione

- se $r = 0$ e le due cifre sono $(4, 6)$, scrivi 0, poni $r = 1$ e spostati a sinistra

  diventa

  $$\langle q_0,\ (4,6,\square),\ (4,6,0),\ q_1,\ \mathrm{sx}\rangle$$

  Poiche specifica 2 condizioni e 3 azioni, l'istruzione prende il nome di **quintupla**, quelli che abbiamo chiamato "stati interiori" si chiamano **stati interni** e l'esecuzione delle quintuple su un insieme di dati si dice **computazione**.

Lezione 2 - macchine di Turing

****

Per risolvere un problema π è necessario progettare una macchina di Turing T_π apposita per risolvere quel problema. Una volta eseguite le istruzioni codificate nelle quintuple di T_π su un dato input x, otteniamo una soluzione per l'istanza x di π.
Una macchina di Turing non è altro che un algoritmo, quindi il *modello di calcolo* **M**acchina di Turing è un linguaggio utilizzato per descrivere gli algoritmi.

macchine di Turing

Def. Macchina di Turing
Sia Σ un *alfabeto* finito e Q un insieme finito di *stati*, in cui distinguiamo lo stato iniziale q_0 ed un insieme di stati finali Q_F. Una **Macchina di Turing** T sul'alfabeto Σ e sull'insieme di stati Q è un dispositivo di calcolo dotato di:

- un'unità di controllo che ad ogni istante si trova in uno stato qualsiasi di Q
- un nastro di lettura/scrittura di lunghezza infinita suddivisa in celle contenenti i caratteri di Σ oppure il carattere $\square$ (blank).
- una testina di lettura/scrittura posizionata sulla cella del nastro
- un programma, ossia un insieme P di quintuple del tipo $\langle\ q_1, a_1, a_2, q_2, m\rangle$ in cui $q_1 \in Q - Q_F$, $q_2 \in Q$, $a_1, a_2 \in \Sigma$, e $m \in \{\text{sinistra, destra, fermo}\}$

Il funzionamento è semplice: quando l'unità di controllo si trova in uno stato q_i, la testina legge il simbolo contenuto nella testina su cui è posizionato, cerca una quintupla che ha come primi elementi q_i e il simbolo letto e, se la trova, esegue le tre azioni in essa indicate, ovvero

- sovrascrive il simbolo nella cella puntata con il simbolo indicato nella quintupla
- cambia (eventualmente) lo stato interno
- muove (eventualmente) la testina
  Eseguita la prima quintupla, si cerca un'altra quintupla da eseguire, finché nessuna quintupla può essere eseguita.

Una **macchina di Turing** quindi è una **quintupla** $\langle\Sigma, Q, q_0, Q_F, P\rangle$

Def. macchina di Turing a k nastri
Una macchina di Turing a k nastri è caratterizzata da:

- un alfabeto Σ
- un insieme *finito* Q di **stati interni**
- uno stato interno iniziale
- un sottoinsieme Q_F di Q di **stati finali**
- un insieme P di **quintuple** che hanno la forma

$$\langle q_1, (a_1, a_2, \ldots, a_k), (b_1, b_2, \ldots, b_k), q_2, (m_1, m_2, \ldots, m_k) \rangle$$

  - dove a_1 è il carattere letto sul nastro 1, a_2 è il carattere letto sul nastro 2 ecc.
  - b_1 è il carattere scritto sul nastro 1, b_2 è il carattere scritto sul nastro 2 ecc.
  - m_1 è il movimento da compiere sul nastro 1 ecc.

Oss.
Il numero di componenti del secondo elemento delle quintuple di P corrisponde proprio al numero di nastri della macchina di Turing

Stati globali

Uno stato globale SG di una macchina di Turing è *informalmente* una fotografia della macchina ad un certo istante.
Formalmente uno **stato globale** di una macchina di Turing T ad un certo istante, è una parola che contiene una descrizione della porzione *non blank* del nastro della machina T, della posizione della testina e dallo stato interno. È rappresentato mediante la sequenza di caratteri (non blank) contenuti sul nastro in cui il carattere letto dalla testina *è preceduto* dallo stato interno.

"FI/MOD II/img/img3.png" non trovato.

Nel caso (a) lo stato globale SG è $q_0 = 812 + 53$, mentre nel caso (b) lo stato globale SG è $= 812 + q_3^0 5$

Transizioni

Una transizione dallo stato globale SG_1 allo stato globale SG_2 avviene quando viene eseguita una quintupla che trasforma SG_1 e SG_2.
Se $T = \langle \Sigma, Q, q_0, Q_F, P \rangle$ è una macchina di Turing ad un nastro, esiste una **transizione** da $SG_1 \to SG_2$ se esiste una quintupla $\langle q, x, x', q', m \rangle \in P$ tale che:

- in SG_1, T si trova nello stato interno $q \in Q$
- in SG_1, la testina di T sta leggendo la cella con il carattere $x \in \Sigma$
- in SG_2 la cella che in SG_1 conteneva il carattere x, ora contiene il carattere $x' \in \Sigma$

- in SG_2, T si trov nello stato $q' \in Q$
- in SG_2, la testina di T sta scandendo la cella in posizione m rispetto a quella che stava scandendo in SG_1

"FI/MOD II/img/img4.png" non trovato.

Nella figura vediamo la transizione da $SG_{(a)} \ = 812 + q_3^0 5$ a $SG_{(b)} = 812 q_3^0 + 5$ data dalla quintupla $\langle q_0^3, 5, 5, q_0^3, \text{sx} \rangle$

Computazione

Informalmente, una computazione di una macchina di Turing *deterministica* a un nastro $T = \langle \Sigma, Q, q_0, Q_F, P \rangle$ è l'esecuzione delle quintuple di T su una sequenza di caratteri scritti sul nastro.

Def. Computazione
Una **computazione** di una macchina di Turing T è una sequenza $SG_0, SG_1, \dots, SG_h, \dots$ di stati globali di T tali che:

- SG_0 è uno *stato globale iniziale*, ossia uno stato globale nel quale lo stato interno è q_0 e la testina è posizionata sul carattere più a sinistra del nastro
- per ogni $0 \leq i \leq h-1$, esiste una transizione da $SG_i \rightarrow SG_{i+1}$ oppure per ogni $h \geq i+1$, SG_h non è definito
  Se esiste un indice h tale che SG_h è uno stato globale dal quale non può avvenire alcuna *transizione* allora la **computazione termina** e questo accade quando lo stato interno nel quale T si trova in SG_h è uno stato finale oppure quando P non contiene una quintupla che possa essere eseguita in SG_h

Trasduttori e riconoscitori

Nelle dispense sono presenti diversi tipi di macchine di Turing: quella che *ordina* gli elementi del nastro, quella che *verifica* se una parola è palindroma, quella che esegue la *somma* di due numeri. Mentre per la prima e la terza operazione il risultato è qualcosa che viene scritto sul nastro, per l'operazione di *verifica* ha come risultato lo stato interno in cui termina.
Macchine del primo tipo sono dette **trasduttori** e hanno il compito di *calcolare valori* e lo stato finale non ha importanza per il risultato, per questo l'insieme degli stati finali di una macchina di tipo trasduttore è costituito da un solo elemento, detto q_F. Le macchine di tipo trasduttore hanno sempre un *nastro di output* sul quale viene scritto il valore della funzione da calcolare
Il compito delle macchine di tipo **riconoscitore** è quello di decidere se l'input appartiene ad un

insieme, quindi l'insieme degli stati finali di una macchina riconoscitore è *sempre* costituito da due stati, ovvero $\{q_A, q_R\}$, rispettivamente *stato di accettazione* q_A e *stato di rigetto* q_R

Indichiamo con $O_T(x)$ l'esito della computazione $T(x)$ della macchina T sull'input x.

> **Esito della computazione per macchine trasduttori**
> Sia T una macchina di Turing di tipo trasduttore; la funzione $O_T(x) : \Sigma^* \to \Sigma^*$ è definita per i soli $x \in \Sigma^*$ tali che $T(x)$ termina, e per tali x, il valore $O_T(x)$ è la parola calcolata da tale computazione (scritta sul nastro di output).

> **Esito della computazione per macchine riconoscitori**
> Sia T una macchina di Turing di tipo riconoscitore; la funzione $O_T(x) : \Sigma^* \to \{q_A, q_R\}$ è definita per i soli $x \in \Sigma^*$ tali che $T(x)$ termina, e per tali x, il valore $O_T(x)$ è lo stato finale di terminazione della computazione.

Lezione 3 - Equivalenza dei modelli delle macchine di Turing

Introduciamo dei diversi tipi di macchine di Turing:

> **macchine di Turing a testine solidali**
> In ogni istruzione le celle dei nastri scandite dalle testine hanno lo stesso indirizzo, quindi, assumendo che all'inizio della computazione tutte le testine siano posizionate sull'indirizzo 0 dei rispettivi nastri, all'istante successivo **tutte** le testine saranno posizionate all'indirizzo +1 o -1.
>
> Una quintupla di una macchina di Turing a k nastri a testine solidali ha la forma
>
> $$\langle q_i, \overline{s_1}, \overline{s_2}, q_j, mov \rangle$$
>
> dove $\overline{s_1} = (s_{1_1}, s_{1_2}, \dots, s_{1_k})$, $\overline{s_2} = (s_{2_1}, s_{2_2}, \dots, s_{2_k})$ e $mov \in \{\text{destra,sinistra,fermo}\}$. Il significato di una quintupla è la seguente: *se* la macchina è nello stato q_i, la testina 1 legge il simbolo s_{1_1} sul nastro 1, la testina 2 legge il simbolo s_{1_2} sul nastro 2,. . .,la testina k legge il simbolo s_{1_k} sul nastro k, *allora* la testina 1 scrive il simbolo s_{2_1} sul nastro 1,. . ., la testina k scrive il simbolo s_{2_k} sul nastro k, la macchina entra nello stato q_j e tutte le testine eseguono il movimento mov.

> **macchine di Turing a testine indipendenti**
> In una macchina di questo tipo, in seguito all'esecuzione di una quintupla le testine si muovono indipendentemente l'una dalle altre.
>
> Una quintupla di una macchina di Turing a k nastri a testine indipendenti ha la forma
>
> $$\langle q_i, \overline{s_1}, \overline{s_2}, q_j, \overline{mov} \rangle$$
>
> dove $\overline{s_1}, \overline{s_2}$ come sopra e $\overline{mov} = (mov_1, mov_2, \dots, mov_k)$ e $mov_h \in \{\text{sinistra,fermo,destra}\}$ per ogni $h = 1, 2, \dots, k$. Il significato di una quintupla è la seguente: *se* la macchina è nello stato q_i, la testina 1 legge il simbolo s_{1_1} sul nastro 1, la testina 2 legge il simbolo s_{1_2} sul nastro 2,. . .,la testina k legge il simbolo s_{1_k} sul nastro k, *allora* la testina 1 scrive il simbolo s_{2_1} sul nastro 1,. . ., la testina k scrive il simbolo s_{2_k} sul nastro k, la

> macchina entra nello stato q_j e la testina 1 esegue il movimento $mov_1, \ldots$, la testina k esegue il movimento mov_k.

Inoltre possiamo anche prendere in considerazione macchine con **un solo nastro di lettura/scrittura**, macchine che usano un **alfabeto con tanti simboli**, macchine che usano un **alfabeto binario**.
Possiamo dimostrare che tutto quello che possiamo fare con una macchina di un qualsiasi tipo, è possibile farla con macchine di diverso tipo.

Equivalenza testine solidali e indipendenti

Poiché una macchina a testine solidali può essere considerata come una particolare macchina a testine indipendenti in cui, ogni volta che si esegue una quintupla, tutte le testine si muovono allo stesso modo, allora **tutto ciò che facciamo con il modello a testine solidali, possiamo farlo anche con quello a testine indipendenti**.

Mostriamo l'inverso, ovvero che **tutto ciò che facciamo col modello a testine indipendenti riusciamo a farlo col modello a testine solidali**.

Consideriamo una macchina di Turing T_2 a testine indipendenti dotata di 2 nastri e mostriamo come simularne il comportamento mediante una macchina di Turing T_3 a testine solidali. I primi due nastri di T_3 sono *inizialmente* una copia dei nastri di T_2, mentre il terzo nastro contiene un solo carattere non appartenente a Σ, ovvero "$*$" nella cella di indirizzo 0, ed ha lo scopo di segnalare alle testine dove posizionarsi.

La macchina T_3 simula T_2 mediante una *sequenza di shift* dei contenuti dei primi due nastri in modo tale che ad ogni passo, la testina di T_3 sia sempre posizionata sulle celle il cui indirizzo coincide con quella della cella del terzo nastro in cui è scritto il carattere $*$.

Sia $\langle q_1, (a, b), (c, d), q_2, (\text{mov}_1, \text{mov}_2)\rangle$ una quintupla di T_2 e vediamo come dipendentemente dai valori di mov_1 e mov_2.

- se $\text{mov}_1 = \text{mov}_2$ allora la quintupla $\langle q_1, (a, b), (c, d), q_2, (\text{mov}_1)\rangle$= $\langle q_1, (a, b), (c, d), q_2, (\text{mov}_2)\rangle$ è inserita nelle quintuple della macchina di Turing T a testine solidali.

Il caso $\text{mov}_1 \neq \text{mov}_2$

Le cose si complicano quando $\text{mov}_1 \neq \text{mov}_2$.
Sia $\langle q_1, (a, b), (c, d), q_2, (\text{mov}_1, \text{mov}_2)\rangle$ di T_2 a testine indipendenti. Dopo aver scritto i caratteri c e d rispettivamente sul primo e secondo nastro, eseguiamo uno shift dei primi due nastri in modo tale che i caratteri che devono essere letti in seguito si trovino nelle stesse celle aventi lo stesso

indirizzo della cella del terzo nastro in cui è scritto il carattere "$*$".
Consideriamo come esempio il caso in cui $\mathrm{mov}_1 = \mathrm{destra}$ e $\mathrm{mov}_2 = \mathrm{sinistra}$. Per eseguire questi due movimenti, dopo aver letto e riscritto i caratteri, basta *traslare* il contenuto del primo nastro di una cella a sinistra (corrisponde ad uno spostamento della testina a destra nella macchina a testine indipendenti) ed il contenuto del secondo nastro di una cella a destra.
Alla quintupla $\langle q_1, s_{1_1}, s_{1_2}), (s_{2_1}, s_{2_2}), q_2, (\mathrm{destra}, \mathrm{sinistra})\rangle$ di T_2 corrisponde l'insieme di quintuple di T_3:

$\langle q_1, (s_{1_1}, s_{1_2}, *), (s_{2_1}, s_{2_2}, *), q^{DS}_{1,D}(q_2),\ \mathrm{destra}\rangle$

$\langle q^{DS}_{1,D}(q_2), (x, y, z), (x, y, z), q^{DS}_{1,D}(q_2),\ \mathrm{destra}\rangle\ \forall x \in \Sigma, \forall y \in \Sigma \cup \{\Box\}, \forall z \in \{*, \Box\}$
(sposta la testina sull'ultimo carattere non $\Box$ sul primo nastro)

$\langle q^{DS}_{1,D}(q_2), (\Box, y, z), (\Box, y, z), q^{DS}_{1,S}(q_2, \Box), \mathrm{sinistra}\rangle\ \forall x \in \Sigma, \forall y \in \Sigma \cup \{\Box\}, \forall z \in \{*, \Box\}$
(si prepara a spostare ciascun carattere sul primo nastro una cella a sinistra)

$\langle q^{DS}_{1,S}(q_2, a), (x, y, z), (a, y, z), q^{DS}_{1,S}(q_2, x), \mathrm{sinistra}\rangle\ \forall a, x, y \in \Sigma \cup \{\Box\}\ : x \neq \Box\ \vee\ y \neq \Box$
(esegue lo spostamento di ciascun carattere sul primo nastro una cella a sinistra)

$\langle q^{DS}_{2,S}(q_2, a), (\Box, \Box, z), (a, \Box, z), q^{DS}_{2,D}(q_2, \Box), \mathrm{destra}\rangle\ \forall a \in \Sigma \cup \{\Box\}, \forall z \in \{*, \Box\}$
(raggiunta la prima posizione a destra in cui le celle del primo e secondo nastro contengono $\Box$, si prepara a spostare ciascun carattere sul secondo nastro una cella a destra)

$\langle q^{DS}_{2,D}(q_2, b), (x, y, z), (x, b, z), q^{DS}_{2,D}(q_2, y), \mathrm{destra}\rangle\ \forall b, x \in \Sigma \cup \{\Box\},\ \forall y \in \Sigma, \forall z \in \{*, \Box\}$
(esegue lo spostamento di ciascun carattere sul secondo nastro una cella a destra)

$\langle q^{DS}_{2,D}(q_2, b), (x, \Box, z), (x, b, z), q^{DS}_{2,S}(q_2), \mathrm{fermo}\rangle\ \forall b, x \in \Sigma \cup \{\Box\}, \forall z \in \{*, \Box\}$
(terminato lo spostamento di ciascun carattere sul secondo nastro una cella a destra, si prepara a tornare sulla cella del terzo nastro che contiene *)

$\langle q^{DS}_{2,S}(q_2), (x, y, \Box), (x, b, \Box), q^{DS}_{2,S}(q_2), \mathrm{sinistra}\rangle\ \forall x, y, \in \Sigma \cup \{\Box\}$
(raggiunge la cella del nastro che contiene *)

$\langle q^{DS}_{2,S}(q_2), (x, y, *), (x, b,), q_2, \mathrm{fermo}\rangle\ \forall x, y \in \Sigma \cup \{\Box\}$
(entra nello stato q_2: la simulazione dell'esecuzione della quintupla di T_2 è terminata)

La simulazione di una macchina T_k a $k > 2$ nastri a testine indipendenti, viene eseguita tramite una macchina T_{k+1} a $k+1$ nastri a testine solidali, in cui il nastro $k+1$ viene utilizzato per rappresentare la posizione della testina e gli altri k nastri rimanenti vengono shiftati a sinistra o a destra in base al movimento nelle quintuple delle testine indipendenti.

Per dimostrare questa equivalenza abbiamo usato la tecnica della **simulazione**, ovvero quella di progettare una macchina T' con certe caratteristiche che fa la stessa cosa di una macchina T che ha altre caratteristiche.

Macchine a k nastri e macchine ad 1 nastro

Sempre grazie alla tecnica della simulazione, dimostreremo che la capacità computazionale di una macchina di Turing non dipende dal numero di nastri. Quindi dimostreremo che una macchina di Turing a k nastri può essere simulata mediante una macchina di Turing T_1 ad un nastro.

In base a quanto visto nel paragrafo precedente, ci limitiamo a considerare una macchina di Turing a k **testine solidali**.

Sia T_k una macchina di Turing a k nastri a testine solidali. Definiamo una macchina di Turing T_1 ad un nastro che utilizza lo stesso alfabeto Σ utilizzato da T_k ed il cui insieme di stati è $Q \times \Sigma^k$. Inizialmente l'input $x = (x_{1_1}, x_{1_2}, \dots, x_{1_k})(x_{2_1}, x_{2_2}, \dots, x_{2_k}) \dots (x_{n_1}, x_{n_2}, \dots, x_{n_k})$ di T_k è scritto sull'unico nastro di T_1 come concatenazione di tutti i simboli di x, ovvero nella seguente forma: $x_{1_1} x_{1_2} \dots x_{1_k} x_{2_1} x_{2_2} \dots x_{2_k} \dots x_{n_1} x_{n_2} \dots x_{n_k}$.

Sia $\langle q_1, (s_{1_1}, s_{1_2}, \dots, s_{1_k}), (s_{2_1}, s_{2_2}, \dots, s_{2_k}), q_2, m \rangle$ una quintupla di T_k e trasformiamola in un inseme di quintuple per T_1

1. Osserviamo che mentre a T_k è sufficiente una singola operazione di lettura per poter eseguire correttamente la quintupla, T_1 deve eseguire k operazioni di lettura consecutive, in quanto la quintupla viene eseguita solo se viene letto s_{1_1} ed esso è seguito da s_{1_2} e così via fino a s_{1_k}. Quindi per poter eseguire la quintupla è necessario leggere e ricordare k simboli consecutivi

$$
\begin{array}{l}
\langle q_1, s_{1_1}, s_{1_1}, q(q_1, s_{1_1}, \text{destra}) \rangle \\
\langle q(q_1, s_{1_1}), s_{1_2}, s_{1_2}, q(q_1, s_{1_1}, s_{1_2}), \text{destra} \rangle \\
\vdots \\
\langle q(q_1, s_{1_1}, s_{1_2}, \dots, s_{1_{k-1}}), s_{1_k}, s_{1_k}, q(q_1, s_{1_1}, s_{1_2}, \dots, s_{1_{k-1}}, s_{1_k}), \text{sinistra} \rangle
\end{array}
$$

2. Ora, T_1 ha verificato che la quintupla può essere eseguita e, per poterlo fare, deve riportare la testina a sinistra di k posizioni

$$\langle q(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_{k-1}}, s_{1_k}), s_{1_{k-1}}, s_{1_{k-1}}, q(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_{k-1}}, s_{1_k}, k-2), \text{sinistra}\rangle$$
$$\langle q(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_k}, i), s_{1_i}, s_{1_i}, q(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_k}, i-1), \text{sinistra}\rangle \ \forall i = 2, \ldots, k-2$$

3. La testina di T_1 è posizionata sul carattere corrispondente al carattere scritto sul primo nastro di T_k (s_1) e può procedere all'esecuzione della quintupla sovrascrivendo i k caratteri

$$\langle q(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_{k-1}}, s_{1_k}, 1), s_{1_1}, s_{2_1}, q^{\text{write}}(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_{k-1}}, s_{1_k}, 2), \text{destra}\rangle$$
$$\langle q^{\text{write}}(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_k}, i), s_{1_i}, s_{2_i}, q^{\text{write}}(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_k}, i+1), \text{destra}\rangle \forall i = 2, \ldots, k$$
$$\langle q^{\text{write}}(q_1, s_{1_1}, s_{1_2}, \ldots, s_{1_{k-1}}, s_k, k), s_{1_k}, s_{2_k}, q', m'\rangle$$
dove q' e m' dipendono dal valore di m come vedremo in seguito

4. T_1 ha eseguito la prima parte della quintupla, scrivendo i k nuovi caratteri sul nastro; in questo istante, la sua testina è posizionata sulla cella contenente l'ultimo simbolo scritto. Restano da eseguire il cambio di stato interno ed il movimento della testina. Queste operazioni avvengono in maniera differente, dipendentemente dal valore di m
   - Se $m = \text{destra}$, allora $q' = q_2$ e $m' = \text{destra}$ e l'esecuzione della quintupla è terminata
   - Se $m = \text{fermo}$, allora $q' = q^{sin}(q_2, k-1)$ e $m' = \text{sinistra}$ e l'esecuzione della quintupla di T_k termina con le quintuple seguenti:

$$\langle q^{\text{sin}}(q_2, i), x, x, q^{\text{sin}}(q_2, i-1), \text{sinistra}\rangle \ \forall x \in \Sigma \cup \{\square\} \ \forall i = 2, \ldots, k-1$$
$$\langle q^{\text{sin}}(q_2, i), x, x, q_2, \text{fermo}\rangle \ \forall x \in \Sigma \cup \{\square\}$$

   - Se $m = \text{sinistra}$, allora $q' = q^{sin}(q_2, 2k-1)$ e $m' = \text{sinistra}$ e l'esecuzione della quintupla di T_k termina con le quintuple seguenti:

$$\langle q^{\text{sin}}(q_2, i), x, x, q^{\text{sin}}(q_2, i-1), \text{sinistra}\rangle \ \forall x \in \Sigma \cup \{\square\} \ \forall i = 2, \ldots, 2k-1$$
$$\langle q^{\text{sin}}(q_2, i), x, x, q_2, \text{fermo}\rangle \ \forall x \in \Sigma \cup \{\square\}$$

> **Oss.**
> L'insieme degli stati di T_1 ha una cardinalità molto maggiore dell'insieme degli stati di T_k e l'insieme degli stati finali di T_1 coincide con quello degli stati finali di T_k

Riduciamo la cardinalità di Σ

In questo paragrafo vediamo come ogni macchina di Turing T definita su un alfabeto Σ con $|\Sigma| > 2$ possa essere simulata da una macchina di Turing T_{01} definita sull'alfabeto $\{0, 1\}$.
Indichiamo con Q l'insieme degli stati di T e con P l'insieme delle quintuple.
Sia $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, possiamo codificare σ_i con la rappresentazione binaria dell'intero

$i-1$ utilizzando $k = \lceil \log_2 n \rceil$ bit e rappresentiamo con $b(\sigma)$ questa rappresentazione per ogni $\sigma \in \Sigma$.

> **Esempio**
> Se $n = 10$ allora $k = 4$ e $b(\sigma_1) = 0000$, $b(\sigma_2) = 0001, \dots, b(\sigma_9) = 1000, b(\sigma_{10}) = 1001.$

Per ogni elemento $\sigma \in \Sigma$ e per ogni h compreso tra 1 e $k = \lceil \log_2 n \rceil$, indichiamo con $b_h(\sigma)$ l'h-esimo bit di $b(\sigma)$

> **Esempio**
> $b(\sigma_6) = 0101$ e $b_1(\sigma_6) = 0$

Vediamo come simulare il comportamento di T utilizzando una macchina con alfabeto binario T_{01}. Costruiamo T_{01} come una macchina a k nastri e cominciamo a scrivere su ogni nastro la codifica in binaria dell'input scritto sull'unico nastro di T.

- Sia $x_1 x_2 x_3 \dots x_k$ l'input di T (con $x_1 x_2 x_3 \dots x_k$ elementi di Σ)
- Nelle celle di indirizzo 1 dei nastri di T_{01} scriviamo i simboli binari della codifica del carattere x_1, quindi se $b(x_1) = b_1(x_1) b_2(x_2) \dots b_k(x_1)$ (ossia $b_i(x_1)$ indica l'i-esimo bit di $b(x_1)$) allora scriviamo $b_i(x_1)$ nella cella numero 1 dell'i-esimo nastro.

> **Esempio**
> Sia $\Sigma_T = \{a, b, c, d, e, f, g\}$ e $b(a) = 000, b(b) = 001, b(c) = 010, \dots, b(z) = 111$
>
> “FI/MOD II/img/img5.png” non trovato.
>
> A questo punto la quintupla di $\langle q_1, a, c, q_2, m \rangle$ di T diventa la quintupla $\langle q_1, (b_1(a), b_2(a) \dots b_k(a)), (b_1(c) b_2(c) \dots b_k(c)), q_2, m \rangle$ di T_{01}

Grazie a quanto visto nel paragrafo precedente possiamo trasformare la macchina T_{01} a k nastri, in una macchina ad un solo nastro.

Abbiamo visto quindi come costruire una macchina che fa le stesse cose di un'altra, ovvero formalmente, si dice che "**l'esito della computazione di una macchina su un input coincide con l'esito della computazione dell'altra macchina sullo stesso input**".

Lezione 4 - Struttura di P e macchine non deterministiche

Struttura di P

Fino ad ora, non abbiamo posto limitazioni sulla struttura dell'insieme delle produzioni P che definisce una particolare macchina di Turing.
Prendiamo una macchina di Turing, ovvero una quintupla $T = \langle \Sigma, Q, P, Q_F, m \rangle$ e osserviamo che per sapere tutto di T basta avere l'insieme P perché possiamo ricavare Σ e Q.
Ricordiamo che possiamo vedere P come una funzione che associa ad una coppia $(\text{stato}, \text{simbolo})$ una tripla $(\text{simbolo}, \text{stato}, \text{movimento})$ ovvero

$$P : Q \times \Sigma \to \Sigma \times Q \times \{\text{sinistra}, \text{destra}, \text{fermo}\}$$

Totalità

Una quintupla non è altro che un istruzione al linguaggio associato alla macchina di Turing, ad esempio la quintupla $\langle q_1, a, b, q_2, m \rangle$ ci dice che *se la macchina T si trova nello stato q_1 e leggiamo il carattere a allora scriviamo b, passiamo nello stato q_2 ed eseguiamo il movimento m*.
Ma cosa succede se, trovandoci in uno stato q e leggendo un carattere x, non troviamo in P alcuna quintupla i cui primi due simboli sono q ed s? Non viene eseguita nessuna azione e quindi T interrompe la sua computazione .

Per **totalità** dell'insieme delle quintuple si intende che *per ogni coppia $(q_1, s_1) \in (Q - Q_F) \times \Sigma$ esiste una tripla $(s_2, q_2, m) \in \Sigma \times Q \times \{\text{destra}, \text{sinistra}, \text{fermo}\}$ tale che $\langle q_1, s_1, s_2, q_2, m \rangle \in P$.*
Questa questione è connessa alla verifica delle *precondizioni*, ovvero, quando progettiamo una macchina di Turing assumiamo che **l'input venga scritto sul nastro in un certo formato**.

> **Esempio**
> Nella progettazione della macchina ad un nastro che calcola la somma di due numeri abbiamo assunto che:
>
> - le cifre del primo numero fossero scritte in celle consecutive

> - la cifra più significativa a sinistra
> - immediatamente seguite dal $+$ (senza $\square$ intermedi) e che il $+$ fosse immediatamente seguito dalle cifre del secondo numero rispettando le due condizioni sopra
>   In questo caso la parola $1\square + 1$ non è un input valido per la macchina, per questo non abbiamo previsto nessuna quintupla i cui primi due elementi fossero $(q_0, \square)$.

Quindi la corrispondenza P non può essere totale, ovvero, considerando P come un insieme di quintuple, esso non può contenere le quintuple che iniziano con coppie di simboli $(\text{stato\_attuale}, \text{simbolo\_letto})$ che si riferiscono a configurazioni del nastro che non rispettano le *precondizioni*.

Andiamo a chiarire il significato che intendiamo quando ad una coppia $(\text{stato}, \text{simbolo})$ non è associata nessuna quintupla in P, distinguendo tra *trasduttori e riconoscitori*.

Trasduttori

Sia T_t una macchina di tipo *trasduttore* che ad un certo passo della computazione su input $x \in \Sigma^*$, si trovi nello stato q e stia leggendo il carattere s. Nel caso in cui nell'insieme delle quintuple P **non esista** alcuna quintupla che inizia con la coppia (q, s), T_t *non è in grado di produrre l'output corrispondente all'input x scritto sul nastro*. Possiamo affermare quindi che la computazione $T_t(x)$ non produce alcun output. Possiamo considerare una nuova macchina T_t' il cui insieme delle quintuple P' sia l'unione dell'insieme delle quintuple P e dell'insieme delle quintuple aggiuntive

$$\{\langle q, (s, x), (s, x), q, \text{fermo}\rangle \;\; x \in \Sigma\}$$

il cui comportamento è sostanzialmente identico a quello di T_t, ovvero $T_t(x) = T_t'(x)$ per ogni $x \in \Sigma^*$ tale che $T_t(x)$ calcola un output e $T_t'(x)$ **non termina** per ogni $x \in \Sigma^*$ tale che $T_t(x)$ **non calcola un output** (osservare la differenza tra non termina e non calcola l'output).

Riconoscitori

Consideriamo una macchina di Turing T di tipo riconoscitore e supponiamo che essa si trovi nello stato q, che la testina legga il simbolo s e che nell'insieme delle sue quintuple P non esista nessuna quintupla che inizi con la coppia (q, s); in questo caso la macchina T non riuscirà a raggiungere uno stato di accettazione. Possiamo quindi aggiungere all'insieme P la quintupla

$$\langle q, s, s, q_R, \text{fermo}\rangle$$

senza alterare l'insieme delle parole accettate da T.

P è una corrispondenza o una funzione?

Abbiamo detto che l'insieme delle quintuple P è una *corrispondenza* fra $Q \times \Sigma$ e $\Sigma \times Q \times \{\text{destra}, \text{sinistra}, \text{fermo}\}$, ma negli esempi abbiamo visto che la corrispondenza si è rivelata invece una **funzione**, infatti P non conteneva coppie di quintuple che avevano gli stessi due elementi iniziali. Questo corrisponde quindi ad una sequenza di istruzioni che **devono** essere eseguite ogni volta che si verifica una condizione, quindi la macchina non ha gradi di libertà, si dice che la macchina di Turing è **deterministica**.

È stato definito anche il modello di macchina di Turing *non deterministica*, ossia quella il cui insieme P può contenere più quintuple che iniziano con la stessa coppia stato-carattere. Il *grado di non determinismo* è il numero massimo di quintuple che iniziano con la stessa coppia stato-carattere.

> **Oss.**
> Il grado di non determinismo di una macchina di Turing è al più $3|Q| \cdot |\Sigma|$

Una macchina non deterministica ha quindi l'insieme delle quintuple che potrebbe avere una struttura simile

$$\langle q, a, b_1, q_1, m_1 \rangle \ \langle q, a, b_2, q_2, m_2 \rangle \ \ \ldots \ \langle q, a, b_k, q_k, m_k \rangle$$

e chiamiamo questa struttura una *multi-quintupla*.

Cosa accade quindi quando l'insieme di quintuple di una macchina di Turing T ha la multi-quintupla presentata sopra e durante la sua computazione $T(x)$ si trova nello stato interno q e legge il carattere a?

Una computazione non deterministica può essere vista come un albero sorgente (orientato dalla radice alle foglie) i cui nodi sono gli stati globali utilizzati dalla computazione. La radice dell'albero è lo stato globale iniziale della computazione e i figli di ciascun nodo interno sono gli stati globali che corrispondono alla esecuzione di una istruzione non deterministica. Quindi se k è il grado di non determinismo di una macchina di Turing NT, ogni nodo dell'albero della computazione ha al più k figli.

Chiamiamo **computazione deterministica di $NT(x)$** ciascun percorso nell'albero uscente dalla radice e che termina solo quando incontra una foglia.

Sia NT una macchina di Turing non deterministica, siano q_A e q_R gli stati di accettazione e rigetto di NT e sia x l'input di NT, allora

$$O_{NT}(x) = \begin{cases} q_A & \text{almeno una delle computazioni deterministica di } NT(x) \text{ termina in } q_A \\ q_R & \text{tutte le computazioni deterministiche di NT(x) terminano in } q_R \\ \text{non definito} & \text{altrimenti} \end{cases}$$

"FI/MOD II/img/img6.png" non trovato.

Teorema 2.1
Per ogni macchina di Turing non deterministica NT esiste una macchina di Turing deterministica T tale che, per ogni input x di NT, l'esito della computazione $NT(x)$ coincide con l'esito della computazione $T(x)$.

Dim.
È presentata una dimostrazione informale della dimostrazione. Viene applicata la tecnica della simulazione, ossia costruendo una macchina deterministica T che simula il comportamento di una macchina non deterministica NT che ha grado di non determinismo k.
La macchina T su input x esegue una visita dell'albero corrispondente alla computazione $NT(x)$. La visita non può essere in profondità, perché non conosciamo la lunghezza delle computazioni e qualcuna di loro potrebbe non terminare.
Viene dimostrato utilizzando una particolare visita in ampiezza, basata sulla tecnica della *coda di rondine con ripetizioni*.
Partiamo dallo stato globale iniziale $SG(T, x, 0)$ e simuliamo *tutte* le computazioni lunghe un passo (che sono al più k) e se non possiamo concludere nulla sull'esito della computazione $NT(x)$, allora torniamo a $SG(T, x, 0)$ e simuliamo *tutte* le computazioni lunghe due passi (al più k^2) e così via finché non si trova una computazione deterministica accettante.

Lezione 5 - La macchina di Turing Universale

Abbiamo visto che una macchina di Turing è un algoritmo, descritto nel linguaggio delle quintuple, che risolve un problema. L'input, per una macchina di Turing, è una *parola* costituita da caratteri di un certo alfabeto.

Abbiamo detto nella lezione precedente che è sufficiente conoscere l'insieme P per sapere tutto di una macchina di Turing T, ma P non ci dice proprio tutto, infatti dobbiamo conoscere quale sia lo stato iniziale e gli stati finali della macchina.

Quindi data una macchina T, se decidiamo di costruire una parola secondo i seguenti punti

- il primo carattere della parola è q_0 che è seguito dal carattere "-" $\notin \Sigma$
- seguito da q_A poi da "-" e poi da q_R
- e questo seguito, una dopo l'altra da tutte le quintuple di T
  La parola che abbiamo appena costruito *definisce completamente T*.

> **Esempio**
> Prendiamo una macchina T_{PPAL} che termina in q_A se la parola scritta sul suo nastro ha lunghezza pari ed è palindroma.
> Il suo stato iniziale è q_0, lo stato di accettazione è q_A, quello di rigetto q_R e le sue quintuple sono
>
> $$\begin{aligned}&\langle q_0, a, \square, q_a, D\rangle, \langle q_0, b, \square, q_b, D\rangle\\&\langle q_a, a, a, q_a, D\rangle, \langle q_a, b, b, q_a, D\rangle, \langle q_b, a, a, q_b, D\rangle, \langle q_b, b, b, q_b, D\rangle\\&\langle q_a, \square, \square, q_{a1}, S\rangle, \langle q_b, \square, \square, q_{b1}, S\rangle\\&\langle q_2, a, a, q_2, S\rangle, \langle q_2, b, b, q_2, S\rangle, \langle q_2, \square, \square, q_0, S\rangle\\&\langle q_0, \square, \square, q_A, F\rangle\end{aligned}$$
>
> allora T_{PPAL} è completamente descritta dalla seguente parola
>
> $$q_0 - q_A - q_R\langle q_0, a, \square, q_a, D\rangle, \langle q_0, b, \square, q_b, D\rangle\langle q_a, a, a, q_a, D\rangle, \langle q_a, b, b, q_a, D\rangle, \langle q_b, a, a, q_b, D\rangle, \langle q_b, b, b, q_b, D\rangle\langle q_a, \square, \square, q_{a1}, S\rangle, \langle q_b, \square, \square, q_{b1}, S\rangle\langle q_2, a, a, q_2, S\rangle, \langle q_2, b, b, q_2, S\rangle, \langle q_2, \square, \square, q_0, S\rangle\langle q_0, \square, \square, q_A, F\rangle$$
>
> > **Oss.**
> > L'insieme delle quintuple di T_{PPAL} non è una funzione totale, infatti non è considerato il caso

> in cui la parola in input ha lunghezza dispari. In questo caso $T_{PPAL}(x)$ termina:
>
> - nello stato q_{a1} se x è una parola palindroma di lunghezza dispari ed ha a al centro
> - nello stato q_{b1} se x è una parola palindroma di lunghezza dispari ed ha b al centro
>   Possiamo completare P aggiungendo le quintuple
>
> $$\langle q_{a1}, \square, \square, q_R, F\rangle, \langle q_{b1}, \square, \square, q_R, F\rangle$$
>
>   In questo modo, poiché vogliamo che T_{PPAL} termini in q_A solo se la parola scritta sul suo nastro ha lunghezza pari ed è palindroma, allora T_{PPAL} rigetta le parole palindrome di lunghezza dispari

Da tutto quello che abbiamo visto sulle parole, possiamo quindi affermare che una *macchina di Turing è una parola* costituita dai caratteri dell'alfabeto $Q \cup \Sigma \cup \{-, \langle, \rangle, \square\}$.
Quindi, siccome è una parola, allora possiamo scriverla sul nastro di un'altra macchina di Turing, diciamo U, così che lavori sulla nostra macchina di Turing come input.
Se sul nastro di U ci scriviamo, oltre alla parola che descrive la nostra macchina di Turing di partenza (chiamiamola T), anche un input x di T, allora U *simula* la computazione di $T(x)$ e se chiamiamo p_T la parola che descrive T, l'esito della computazione $U(p_T, x) = T(x)$.
A cosa serve questo?
Pensiamo se riuscissimo a progettare una macchina di Turing U che prende in input due parole:

- p_T che descrive una **qualsiasi** macchina di Turing T
- una parola x, input di T
  e che riesce a simulare la computazione $T(x)$ per **qualunque macchina di Turing T**.
  Secondo quest'idea U sarebbe una macchina di Turing alla quale potrei comunicare un qualsiasi algoritmo e un input per quell'algoritmo e U eseguirebbe l'algoritmo su quell'input.
  Quindi in sostanza la macchina di Turing U è **l'algoritmo che descrive il comportamento di un calcolatore** e prende il nome di **macchina di Turing Universale**.

La macchina di Turing Universale

Progettiamo la macchina di Turing Universale che, presi in input la descrizione di una macchina di Turing T (ad un nastro) ed un input $x \in \{0, 1\}^*$ di T, esegue una computazione con esito uguale a quello di $T(x)$.
La macchina di Turing Universale U è una macchina che utilizza 4 nastri a testine indipendenti (possiamo poi trasformarla grazie alle nozioni nella lezione 3):

- N_1 il nastro su cui è memorizzata la descrizione di T

- N_2, il nastro di lavoro di U su cui è memorizzato l'input x della macchina di Turing T che vogliamo simulare
- N_3 il nastro su cui, ad ogni istante della computazione che simula $T(x)$ sarà memorizzato lo stato attuale della macchina T
- N_4, il nastro su cui viene scritto lo stato di accettazione della macchina T

"FI/MOD II/img/img7.png" non trovato.

Assumiamo che T sia descritta dalla parola $\rho_T \in [Q_t \cup \{0, 1, \oplus, \otimes, -\}]^*$ seguente

$$\rho_T = \omega_0 - \omega_1 \otimes \omega_{1_1} - b_{1_1} - b_{1_2} - \omega_{1_2} - m_1 \oplus \cdots \oplus \omega_{h_1} - b_{h_1} - b_{h_2} - \omega_{h_2} - m_h \oplus$$

dove ω_0 e ω_1 indicano rispettivamente stato iniziale e di accettazione di T.

La macchina U per simulare il comportamento della macchina T esegue il seguente algoritmo

Algoritmo di esecuzione della mdT Universale

1. Nello stato q_0, vengono copiati ω_0 sul nastro N_3 e ω_1 su N_4. La testina di N_1 viene spostata sul simbolo a destra del primo carattere $\otimes$ che incontra e la macchina entra nello stato q_1

$$\begin{array}{lr}
\langle q_0, (x, a, \square, \square), (x, a, x, \square), q_0, (d, f, f, f)\rangle & \forall x \in Q_T \wedge \forall a \in \{0, 1, \square\} \\
\langle q_0, (-, a, x, \square), (-, a, x, \square), q_0, (d, f, f, f)\rangle & \forall x \in Q_T \wedge \forall a \in \{0, 1, \square\} \\
\langle q_0, (y, a, x, \square), (y, a, x, y), q_0, (d, f, f, f)\rangle & \forall x, y \in Q_T \wedge \forall a \in \{0, 1, \square\} \\
\langle q_0, (\otimes, a, x, y), (\otimes, a, x, y), q_1, (d, f, f, f)\rangle & \forall x, y \in Q_T \wedge \forall a \in \{0, 1, \square\}
\end{array}$$

2. Nello stato q_1 inizia la ricerca di una quintupla su N_1 che abbia come primo simbolo lo stesso simbolo che si legge dalla testina su N_3 e come secondo simbolo lo stesso simbolo letto dalla testina di N_2
   - se nello stato q_1 legge lo stesso simbolo sui nastri N_1 e N_3, sposta la testina su N_1 a destra di due posizioni ed entra nello stato $q_{\text{StatoCorretto}}$

$$\begin{array}{lr}
\langle q_1, (x, a, x, y), (x, a, x, y), q_1, (d, f, f, f)\rangle & \forall x, y \in Q_T \wedge \forall a \in \{0, 1, \square\} \\
\langle q_1, (-, a, x, y), (-, a, x, y), q_{\text{StatoCorretto}}, (d, f, f, f)\rangle & \forall x, y \in Q_T \wedge \forall a \in \{0, 1, \square\}
\end{array}$$

   Ora la testina N_1 è posizionata sul secondo elemento, ovvero il carattere letto della quintupla che si sta esaminando
   1. Se nello stato $q_{\text{StatoCorretto}}$ legge lo stesso simbolo sui nastri N_1 e N_2, allora ha trovato la quintupla da eseguire, quindi sposta N_1 a destra di due posizioni ed entra nello stato q_{scrivi}
   2. Se nello stato $q_{\text{StatoCorretto}}$ legge simboli diversi sui nastri N_1 e N_2, allora la quintupla che

sta leggendo su N_1 non è quella corretta, quindi entra nello stato q_2 e sposta la testina su N_1 sul primo simbolo successivo al primo $\oplus$ che incontra, e se questo simbolo non è $\square$ entra in q_1 , altrimenti entra nello stato di rigetto
- se nello stato q_1 legge simboli differenti su N_1 e N_3 allora la quintupla che stiamo scandendo non è quella da eseguire (lo stato da cui parte è diverso), quindi entra nello stato q_3 e sposta la testina di N_1 sul primo simbolo successivo al primo $\oplus$ che incontra e se questo simbolo non è $\square$ entra nello stato q_1, altrimenti confronta lo stato attuale che legge su N_3 e lo confronta con lo stato di accettazione ω_1 scritto su N_4 e se sono uguali entra nello stato di accettazione, altrimenti rigetta.

3. Nello stato q_{scrivi} inizia l'esecuzione della quintupla che ha individuato sul nastro N_1 scrivendo il nuovo simbolo su N_2, che legge sul nastro N_1, poi entra nello stato $q_{\text{CambiaStato}}$ spostandosi a destra di due posizioni
4. Nello stato $q_{\text{cambiaStato}}$ prosegue l'esecuzione della quintupla individuata sul nastro N_1 modificando il contenuto del nastro N_3 scrivendoci lo stato che legge sul nastro N_1 ed entra nello stato q_{muovi} muovendo due posizioni a destra la testina su N_1
5. Nello stato q_{muovi} termina l'esecuzione della quintupla letta sul nastro N_1, eseguendo il movimento letto sul nastro N_2 e la macchina entra nello stato $q_{\text{riavvolgi}}$
6. Nello stato $q_{\text{riavvolgi}}$ viene riposizionata la testina di N_1 sul primo simbolo a destra del carattere $\otimes$ ed entra nello stato $q_1$eseguendo

> **Oss.**
> La computazione $U(p_T, x)$ rigetta ogni volta che U non trova una quintupla da eseguire e lo stato scritto su N_3 non è uguale allo stato scritto su N_4, quindi U rigetta il suo input (p_T, x) senza verificare che la computazione $T(x)$ abbia rigettato.

Osserviamo però che vogliamo una macchina U che sappia simulare **qualsiasi** macchina di Turing T e nella descrizione che abbiamo dato prima, la macchina utilizza l'insieme degli stati Q_T come alfabeto della macchina U, ma ogni macchina T ha un suo insieme degli stati Q e un suo alfabeto Σ, quindi la prima cosa che potremmo pensare è che U deve avere un alfabeto infinito, cosa non possibile in quanto sappiamo *che l'alfabeto di una macchina di Turing deve avere cardinalità costante*.

La soluzione è quella di codificare tutto in binario, quindi abbiamo l'alfabeto $\Sigma_U = \{0, 1, \otimes, \oplus, -, f, s, d\}$ e definiamo $b^Q : Q \to \lceil \log |Q| \rceil$ una funzione che codifica in binario gli stati di T utilizzando per ciascuno di essi $m = \lceil \log |Q| \rceil$ cifre e per ogni $\omega \in Q$ indichiamo con $b^Q(\omega) = b_1^Q(\omega) b_2^Q(\omega) \dots b_m^Q(\omega)$ la codifica di ω.
Secondo questa soluzione, allora, la descrizione di T è la parola $\beta_T \in \Sigma^*$ descritta in questo modo

$$\beta_T = b^Q(w_0) - b^Q(w_1) \otimes b^Q(w_{1_1}) - b_{1_1} - b_{1_2} - b^Q(\omega_{1_2}) - m_1 \oplus \cdots \oplus$$

Algoritmo di esecuzione raffinato

Nella descrizione dell'algoritmo precedente, bisogna quindi raffinare la descrizione delle operazioni, che nella descrizione precedente erano atomiche, mentre ora diventano operazioni da eseguire mediante cicli.

1. A partire dallo stato q_0 vengono copiati gli m caratteri della codifica $b^Q(\omega_0)$ sul nastro N_3 e gli m caratteri della codifica $b_Q(\omega_1)$ sul nastro N_4 e le testine di N_3 ed N_4 vengono spostate a sinistra sul primo carattere scritto, mentre la testina di N_1 viene spostata sul simbolo a destra del simbolo $\otimes$ entrando nello stato q_1

$$\begin{array}{lr}
\langle q_0, (x, a, \square, \square), (x, a, x, \square), q_0, (d, f, d, f)\rangle & \forall x \in \{0,1\} \wedge \forall a \in \{0,1,\square\} \\
\langle q_0, (-, a, \square, \square), (-, a, \square, \square), q_{01}, (d, f, f, f)\rangle & \forall a \in \{0,1,\square\} \\
\langle q_{01}, (y, a, \square, \square), (y, a, \square, y), q_{01}, (d, f, f, d)\rangle & \forall y \in \{0,1\} \wedge \forall a \in \{0,1,\square\} \\
\langle q_{01}, (\otimes, a, \square, \square), (\otimes, a, \square, \square), q_{02}, (d, f, s, s)\rangle & \forall a \in \{0,1,\square\} \\
\langle q_{02}, (b, a, x, y), (x, a, y, z), q_{02}, (f, f, s, s)\rangle & \forall x, y \in \{0,1\} \wedge \forall a, b \in \{0,1,\square\} \\
\langle q_{02}, (b, a, \square, \square), (z, a, \square, \square), q_1, (f, f, d, d)\rangle & \forall a, b \in \{0,1,\square\}
\end{array}$$

2. Nello stato q_1 inizia la ricerca di una quintupla su N_1 che abbia come primo simbolo la parola scritta su N_3 e come secondo simbolo lo stesso simbolo letto dalla testina su N_2
   - se nello stato q_1 legge la stessa sequenza di simboli su N_1 eN_3 fino a quando non incontra il carattere "$-$" su N_1 e il carattere $\square$ su N_3, allora sposta la testina di N_1 a destra di due posizioni, la testina di N_3 a sinistra di m posizioni ed entra nello stato $q_\text{StatoCorretto}$

$$\begin{array}{lr}
\langle q_1, (x, a, x, y), (x, a, x, y), q_1, (d, f, d, f)\rangle & \forall x, y \in \{0,1\} \wedge \forall a \in \{0,1,\square\} \\
\langle q_1, (-, a, \square, y), (-, a, \square, y), q_{11}, (d, f, s, f)\rangle & \forall y \in \{0,1\} \wedge \forall a \in \{0,1,\square\} \\
\langle q_{11}, (b, a, x, y), (b, a, x, y), q_{11}, (f, f, s, f)\rangle & \forall x, y \in \{0,1\} \wedge \forall a, b \in \{0,1,\square\} \\
\langle q_{11}, (b, a, \square, y), (b, a, \square, y), q_\text{statoCorretto}, (f, f, d, f)\rangle & \forall y \in \{0,1\} \wedge \forall a, b \in \{0,1,\square\}
\end{array}$$

```
        1. Se nello stato $q_\text{statoCorretto}$ legge lo stesso
simbolo sui nastri $N_1$ e $N_2$, allora ha trovato la quintupla da
eseguire, quindi sposta la testina di $N_1$ a destra di due posizioni ed
entra in $q_\text{scrivi}$
```

$$\langle q_\text{statoCorretto}, (a,a,x,y), (a,a,x,y), q'_\text{statoCorretto}, (d,f,f,f)\rangle$$
$$\langle q'_\text{statoCorretto}, (-,a,x,y), (-,a,x,y), q_\text{scrivi}, (d,f,f,f)\rangle$$

```
        2. Se nello stato $q_\text{statoCorretto}$ legge simboli differenti
sui nastri $N_1$ e $N_2$, allora la quintupla che sta scandendo su $N_1$ non
è quella corretta, quindi entra in $q_2$, sposta la testina di $N_1$ a
destra fino a posizionarla sul primo simbolo successivo al carattere
$\oplus$ e se tale simbolo è $0$ oppure $1$ allora entra in $q_1$,
altrimenti rigetta
```

$$\begin{array}{lr}
\langle q_\text{statoCorretto}, (b,a,x,y), (b,a,x,y), q_2, (d,f,f,f)\rangle & \forall x,y \in \{0,1\} \land \forall a,b \in \{0,1,\square\} : a \neq b \\
\langle q_2, (z,a,x,y), (z,a,x,y), q_2, (d,f,f,f)\rangle & \forall a \in \{0,1,\square\} \land \forall z \in \{0,1,-\} \\
\langle q_2, (\oplus,a,x,y), (\oplus,a,x,y), q_{21}, (d,f,f,f)\rangle & \forall x,y \in \{0,1\} \land \forall a \in \{0,1,\square\} \\
\langle q_{21}, (z,a,x,y), (z,a,x,y), q_1, (f,f,f,f)\rangle & \forall x,y,z \in \{0,1\} \land \forall a \in \{0,1,\square\} \\
\langle q_{21}, (z,a,x,y), (z,a,x,y), q_R, (f,f,f,f)\rangle & \forall a \in \{0,1,\square\} \land \forall z \notin \{0,1\}
\end{array}$$

```
- se nello stato $q_{1}$ legge simboli differenti su $N_1$ e $N_3$, allora
la quintupla su $N_1$ è quella sbagliata, quindi entra in $q_3$ e sposta la
testina su $N_3$ a sinistra fino a posizionarla sul primo simbolo non
$\square$, sposta la testina di $N_{1}$ a destra fino a posizionarla sul
primo simbolo successivo a $\oplus$ e, se questo è $0$ oppure $1$, allora
entra in $q_{1}$, altrimenti confronta lo stato attuale che sta leggendo su
$N_3$ con lo stato di accettazione su $N_4$ e se sono uguali entra nello
stato di accettazione, altrimenti rigetta
```

$$\begin{array}{lr}
\langle q_1, (z,a,x,y), (z,a,x,y), q_3, (f,f,s,f)\rangle & \forall x,y,z \in \{0,1\} : z \neq x \\
\langle q_3, (z,a,x,y), (z,a,x,y), q_3, (f,f,s,f)\rangle & \forall x,y,z \in \{0,1\} \\
\langle q_3, (z,a,\square,y), (z,a,\square,y), q_{31}, (f,f,d,f)\rangle & \forall y,z \in \{0,1\} \\
\langle q_{31}, (z,a,x,y), (z,a,x,y), q_{31}, (d,f,f,f)\rangle & \forall x,y \in \{0,1\} \land \forall z \in \Sigma - \{\oplus\} \\
\langle q_{31}, (\oplus,a,x,y), (\oplus,a,x,y), q_{32}, (d,f,f,f)\rangle & \forall x,y \in \{0,1\} \\
\langle q_{32}, (z,a,x,y), (z,a,x,y), q_1, (f,f,f,f)\rangle & \forall x,y,z \in \{0,1\} \\
\langle q_{32}, (z,a,x,y), (z,a,x,y), q_{33}, (f,f,f,f)\rangle & \forall x,y \in \{0,1\} \land \forall z \notin \{0,1\} \\
\langle q_{33}, (z,a,x,x), (z,a,x,x), q_{33}, (f,f,d,d)\rangle & \forall x \in \{0,1\} \land \forall z \in \Sigma \\
\langle q_{33}, (z,a,\square,\square), (z,a,\square,\square), q_A, (f,f,f,f)\rangle & \forall x \in \{0,1\} \land \forall z \in \Sigma \\
\langle q_{33}, (z,a,x,y), (z,a,x,y), q_R, (d,f,f,f)\rangle & \forall x,y \in \{0,1\} : x \neq y \land \forall z \in \Sigma
\end{array}$$

3. Nello stato q_{scrivi} si esegue la quintupla letta sul nastro N_1 scrivendo il nuovo simbolo su N_2 ed entra nello stato $q_{\text{cambiaStato}}$

$$\langle q_{\text{scrivi}}, (b, a, x, y), (b, b, x, y), q'_{\text{scrivi}}, (d, f, f, f)\rangle$$
$$\langle q'_{\text{scrivi}}, (-, a, x, y), (b, b, x, y), q_{\text{cambiaStato}}, (d, f, f, f)\rangle$$

4. Nello stato $q_{\text{cambiaStato}}$ prosegue l'esecuzione della quintupla modificando il contenuto del nastro N_3 sovrascrivendo la sequenza di simboli che legge su N_1 fino a quando non incontra il carattere $-$ su N_1 e il carattere $\square$ su N_3, infine sposta a destra di una posizione (su uno dei caratteri d, s, f che indicano lo spostamento della testina su N_2) sposta a sinistra la testina di N_3 fino a posizionarla a destra del primo $\square$ che incontra ed entra nello stato q_{muovi}

$$\langle q_{\text{cambiaStato}}, (z, a, x, y), (z, a, x, y), q_{\text{cambiaStato}}, (d, f, d, f)\rangle$$
$$\langle q_{\text{cambiaStato}}, (-, a, \square, y), (-, a, \square, y), q_4, (d, f, s, f)\rangle$$
$$\langle q_4, (z, a, x, y), (z, a, x, y), q_4, (f, f, s, f)\rangle$$
$$\langle q_4, (z, a, \square, y), (z, a, \square, y), q_{\text{muovi}}, (f, f, f, f)\rangle$$

5. Nello stato q_{muovi} termina l'esecuzione della quintupla che ha individuato su N_1 muovendo la testina del nastro N_2 ed entra nello stato $q_{\text{riavvolgi}}$ muovendo a sinistra la testina di N_1

$$\langle q_{\text{muovi}}, (s, a, x, y), (s, a, x, y), q_{\text{riavvolgi}}, (f, s, f, f)\rangle$$
$$\langle q_{\text{muovi}}, (f, a, x, y), (f, a, x, y), q_{\text{riavvolgi}}, (f, f, f, f)\rangle$$
$$\langle q_{\text{muovi}}, (d, a, x, y), (d, a, x, y), q_{\text{riavvolgi}}, (f, d, f, f)\rangle$$

6. Nello stato $q_{\text{riavvolgi}}$, riposiziona la testina di N_1 sul primo simbolo a destra del simbolo $\otimes$ e rientra nello stato q_1

$$\langle q_{\text{riavvolgi}}, (z, a, x, y), (z, a, x, y), q_{\text{riavvolgi}}, (s, f, f, f)\rangle$$
$$\langle q_{\text{riavvolgi}}, (\otimes, a, x, y), (\otimes, a, x, y), q_1, (d, f, f, f)\rangle$$

Lezione 6 - Accettabilità, decidibilità e calcolabilità

Abbiamo iniziato il corso cercando di capire come risolvere automaticamente i problemi e abbiamo studiato la soluzione proposta da Turing, che ha introdotto il modello di calcolo Macchina di Turing.
Ora ci domandiamo:

- utilizzando la macchina di Turing possiamo risolvere **tutti** i problemi, oppure esiste qualche problema non risolubile?
- se esiste qualche problema non risolubile con la Macchina di Turing, è possibile risolverlo con un altro modello di calcolo?
  Rispondiamo prima alla seconda domanda

Linguaggi e macchine di Turing

Una macchina di Turing di tipo riconoscitore calcola una funzione booleana, ovvero dato $x \in \Sigma^*$, verifica se x soddisfa una proprietà $\pi(\cdot)$. Se indichiamo con $O_T(x)$ l'esito della computazione $T(x)$, ovvero lo stato raggiunto dalla computazione $T(x)$ nel caso essa termini, allora possiamo dire che

$$\begin{aligned}&O_T(x) = q_A \iff \pi(x)\\ \\ &\text{equiv.}\\ &\{x \in \Sigma^* : O_T(x) = q_A\} = \{x \in \Sigma^* : \pi(x)\}\end{aligned}$$

Quindi possiamo parlare dell'*insieme delle parole accettate da una mdT di tipo riconoscitore*.

> **Def 3.1**
> Un *linguaggio* L è sottoinsieme Σ^* : $L \subseteq \Sigma^*$

> **Def 3.2**
> Il *linguaggio complemento* L^C di un linguaggio $L \subseteq \Sigma^*$ è l'insieme delle parole non contenute in L : $L^C = \Sigma^* - L$

Def 3.3 (Accettabilità)
Un linguaggio $L \subseteq \Sigma^*$ è *accettabile* se esiste una macchina di Turing T tale che

$$\forall x \in \Sigma^*[O_T(x) = q_A \iff x \in L]$$

>>**Oss.**

> La definizione di accettabilità non dice nulla riguardo l'esito della computazione $T(x)$ nel caso in cui $x \notin L$. Se un linguaggio L è accettato da una macchina T e $x \notin L$ potrebbe accadere che $O_T(x) = q_R$ oppure che $T(x)$ non raggiunga mai uno stato finale.
> In poche parole *l'accettabilità di un linguaggio L non dà indicazioni sull'accettabilità del linguaggio L^C*

Def 3.4 (Decidibilità)
Un linguaggio $L \subseteq \Sigma^*$ è *decidibile* se esiste una macchina di Turing T che *termina per ogni input $x\in\Sigma^{*}* }*etaleche$\forall x \in\Sigma^{*}[O_{T}(x)=q_A\iff x\in L]$$

> **Oss.**
> Se una macchina T decide un linguaggio $L \in \Sigma^*$ allora
>
> $$O_T(x) \begin{cases} q_A & \text{se } x \in L \\ q_R & \text{se } x \in L^C \end{cases}$$

Quando un linguaggio L è deciso da una macchina T, scriviamo $L = L(T)$

La differenza tra *decisione e accettazione* di un linguaggio è il comportamento della macchina sul *linguaggio complemento*.

Teorema 3.1
Un linguaggio $L \subseteq \Sigma^*$ è *decidibile* se e solo se L e L^C sono *accettabili*

Dim.
Se L è decidibile allora esiste una macchina di Turing che, per ogni $x \in \Sigma^*$, con input x termina nello stato q_A se $x \in L$ e termina in q_R se $x \in L^C$.
Deriviamo da T una macchina T'. Gli stati di T' sono gli stati di T con l'aggiunta degli stati q'_A e q'_R rispettivamente stato di accettazione e rigetto della macchina T'. Le quintuple di T' sono le stesse quintuple di T con l'aggiunta delle due quintuple

$$\langle q_A, u, u, q'_R, \text{ferma}\rangle \qquad \langle q_R, u, u, q'_A, \text{ferma}\rangle \qquad \forall u \in \Sigma \cup \{\square\}$$

Quindi per ogni $x \in \Sigma^*$, la computazione $T'(x)$ coincide con la computazione $T(x)$ tranne che per l'ultima istruzione eseguita da $T'(x)$: se $T(x)$ termina in q_A allora $T'(x)$ esegue una ulteriore istruzione per entrare nello stato di rigetto, mentre se $T(x)$ termina in q_R allora $T'(x)$ esegue un'ulteriore istruzione che porta nello stato di accettazione.
Quindi T' accetta $x \iff T$ rigetta x, ossia se e solo se $x \in L^C$ e quindi T' accetta L^C.

Vediamo il viceversa. Se L è accettabile e L^C è accettabile, allora esistono due macchine di Turing T_1 e T_2 che per ogni $x \in \Sigma^*$, $T_1(x)$ accetta se e solo se $x \in L$ e $T_2(x)$ accetta se e solo se $x \in L^C$. Ricordiamo che l'esito della computazione non è specificato per la computazione $T_1(x)$ se $x \in L^C$ e per la computazione $T_2(x)$ se $x \in L$.
Definiamo una macchina T che, simulando le computazioni di T_1 e T_2 su input $x \in \Sigma^*$, decide L.

> **Oss.**
> Se T simulasse prima l'intera computazione di T_1 e poi quella di T_2 non ci sarebbe garanzia di terminazione, quindi la simulazione deve avvenire in modo diverso.

T dispone di due nastri, su ciascuno dei quali viene scritto l'input x; la computazione $T(x)$ avviene alternando sui due nastri le singole istruzioni di T_1 e di T_2 nel seguente modo:

1. Esegui una *singola istruzione* di T_1 sul nastro 1: se T_1 entra in uno stato di accettazione allora T **accetta**, altrimenti esegui il passo 2
2. Esegui una *singola istruzione* di T_2 sul nastro 2: se T_2 entra in uno stato di accettazione allora T **rigetta**, altrimenti esegui il passo 1
   Sia ora $x \in \Sigma^*$. Se $x \in L$, allora, prima o poi, al passo 1 T_1 entrerà nello stato di accettazione, portando T ad accettare. Viceversa se $x \in L^C$ allora, prima o poi, al passo 2 T_2 entrerà nello stato di accettazione, portando T a rigettare. Quindi T **decide** L. $\square$

Funzioni calcolabili

Torniamo a considerare macchine di Turing di tipo trasduttore. Assumiamo che le macchine di tipo trasduttore a cui faremo riferimento siano dotate di un nastro di output, che al termine della computazione, contiene il valore di output.

Dato un alfabeto finito Σ, indichiamo con Σ^* l'insieme delle *parole* su Σ, ovvero, l'insieme delle stringhe di lunghezza finita costituite da caratteri in Σ.

> **Def 3.5**
> Siano Σ_1 e Σ_2 due alfabeti finiti; una funzione $f : \Sigma_1^* \to \Sigma_2^*$ è una funzione *calcolabile* se esiste una macchina di Turing T di tipo trasduttore che, dato in input $x \in \Sigma_1^*$ termina con la stringa $f(x)$ scritta sul nastro di output se e solo se $f(x)$ **è definita**.

Osserviamo che la definizione non dice nulla a riguardo delle computazioni $T(x)$ per i quali $f(x)$ non è definita. Il concetto di calcolabilità di una funzione è simile al concetto di accettabilità di un linguaggio, infatti quando scegliamo un input x per il quale $f(x)$ è definita le cose "*vanno bene*", mentre non sappiamo cosa succede per i valori di x per il quale $f(x)$ non è definita.

Sia Σ un alfabeto finito e $L \subseteq \Sigma^*$ un linguaggio. La **funzione caratteristica** $\chi_L : \Sigma^* \to \{0, 1\}$ di L è una funzione totale tale che, per ogni $x \in \Sigma^*$

$$\chi_L(x) = \begin{cases} 1 & \text{se } x \in L \\ \\ 0 & \text{se } x \notin L \end{cases}$$

Teorema 3.2
Un linguaggio L è decidibile se e soltanto se la funzione χ_L è calcolabile

Dim.
Supponiamo che $L \subseteq \Sigma^*$ sia decidibile, allora esiste una macchina di Turing di tipo riconoscitore T con stato di accettazione q_A e stato di rigetto q_R tale che

$$O_T(x) = \begin{cases} q_A & \text{se } x \in L \\ \\ q_R & \text{se } x \notin L \end{cases}$$

Supponiamo che T utilizzi un solo nastro. A partire da T definiamo una macchina di tipo trasduttore T' a 2 nastri che, con input $x \in \Sigma^*$, opera nel seguente modo:

1. sul primo nastro, dove è scritto l'input x, si esegue la computazione $T(x)$
2. se $T(x)$ termina in q_A allora scrive sul nastro di output il valore 1, altrimenti scrive 0. Successivamente termina.

> **Oss.**
> Poiché L è decidibile, il passo 1. termina per ogni input x. Se $x \in L$, allora $T(x)$ termina nello stato di accettazione e quindi al passo 2. $T'(x)$ scrive 1 sul nastro di output. Se $x \notin L$ allora $T(x)$ non termina in accettazione e al passo 2. $T'(x)$ scrive 0 in output. Questo dimostra che χ_L è calcolabile.

Dimostriamo l'inverso. Supponiamo quindi che χ_L sia calcolabile e osserviamo che χ_L è una *funzione totale*. Allora esiste una macchina di Turing T di tipo trasduttore che, per ogni $x \in \Sigma^*$, calcola $\chi_L(x)$. A partire da T, definiamo una macchina di Turing T' di tipo riconoscitore a due nastri che, con input $x \in \Sigma^*$ opera nel seguente modo:

1. sul primo nastro, in cui è scritto l'input x, si esegue la computazione $T(x)$ scrivendo il risultato sul secondo nastro
2. se sul secondo nastro è stato scritto 1 allora la computazione $T'(x)$ termina in accettazione, altrimenti rigetta

> **Oss.**
> Poiché χ_L è totale, il passo 1. termina per ogni input x. Se $\chi_L = 1$, allora il passo 1. termina scrivendo 1 sul secondo nastro e quindi al passo 2. $T'(x)$ termina in accettazione. Se $\chi_L = 0$ allora il passo 1. termina scrivendo 0 sul secondo nastro e $T'(x)$ termina nello stato di rigetto. Questo dimostra che L è decidibile. $\square$

Questo teorema mostra come ad ogni linguaggio decidibile possa essere associata una funzione calcolabile, che è la funzione caratteristica del linguaggio.
Ci domandiamo se è possibile l'inverso, ovvero, data una qualsiasi funzione f, possiamo associargli un linguaggio che sia decidibile se e solo se f è calcolabile?
Iniziamo associando ad ogni funzione $f : \Sigma_1^* \to \Sigma_2^*$ il linguaggio

$$L_f = \{\langle x, y\rangle : x \in \Sigma_1^* \wedge y \in \Sigma_2^* \wedge y = f(x)\}$$

Osserviamo che c'è una differenza tra la decidibilità di un linguaggio e la calcolabilità di una funzione. Mentre un linguaggio è decidibile se esiste un algoritmo che può determinare se una stringa appartiene ad esso, la calcolabilità di una funzione implica che esiste un algoritmo che può calcolare l'output della funzione per ogni possibile input. Tuttavia, il comportamento di tale algoritmo non è definito se l'input non appartiene al dominio della funzione. Per garantire che il linguaggio L_f sia decidibile, è necessario che la funzione f sia totale, cioè restituisca un output per ogni possibile input.

Teorema 3.3
Se $f : \Sigma_1^* \to \Sigma_2^*$ è calcolabile **e totale** allora L_f è decidibile

Dim.
Poiché f è calcolabile e totale, allora esiste una mdT T di tipo trasduttore che, per ogni $x \in \Sigma^*$, calcola $f(x)$. A partire da T definiamo una mdT di tipo riconoscitore T' a due nastri che, con input $\langle x, y\rangle \ : x \in \Sigma_1^* \wedge y \in \Sigma_2^*$ opera nel seguente modo:

1. sul primo nastro viene scritto l'input $\langle x, y\rangle$
2. sul secondo nastro si esegue la computazione $T(x)$ scrivendo il risultato z
3. si esegue un confronto tra z ed y: se $z = y$ allora la computazione $T'(x)$ termina in accettazione, altrimenti rigetta.

> **Oss.**
> Poiché f è totale, il passo 2. termina per ogni input x. Se $f(x) = y$ allora il passo 2. termina scrivendo y sul secondo nastro e al passo 3. $T'(x)$ termina in accettazione. Se invece $f(x) = z \neq y$, allora il passo 2. termina scrivendo z sul secondo nastro e, quindi, al passo 3. $T'(x)$ termina nello stato di rigetto. Questo dimostra la decidibilità di L_f. $\square$

Teorema 3.4
Sia $f : \Sigma_1^* \to \Sigma_2^*$ una funzione. Se L_f è decidibile allora f è calcolabile

Dim.
Poiché $L_f \subseteq \Sigma_1^* \times \Sigma_2^*$ è decidibile, esiste una mdT T di tipo riconoscitore, con stato di accettazione q_A e stato di rigetto q_R, tale che, per ogni $x \in \Sigma_1^*$ e per ogni $y \in \Sigma_2^*$

$$O_T(x, y) = \begin{cases} q_A & \text{se } y = f(x) \\ \\ q_R & \text{altrimenti} \end{cases}$$

Supponiamo che T utilizzi un solo nastro e a partire da T definiamo una macchina di tipo trasduttore T' a 4 nastri, che con input $x \in \Sigma_1^*$ sul primo nastro, opera nel seguente modo:

1. scrive il valore $i = 0$ sul primo nastro
2. enumera tutte le stringhe $y \in \Sigma_2^*$ la cui lunghezza è pari al valore scritto sul primo nastro, simulando per ciascuna di esse la computazione $T(x, y)$, ovvero:
   - sia y la prima stringa di lunghezza i non ancora enumerata, allora scrive y sul secondo nastro
   - sul terzo nastro viene eseguita la computazione $T(x, y)$
   - se $T(x, y)$ termina in q_A allora scrive sul nastro di output la stringa y e termina, altrimenti, incrementando il valore di i, se y era l'ultima stringa di lunghezza i, torna al passo 2.

> **Oss.**
> Poiché L_f è decidibile, il passo 2.2 termina per ogni input $\langle x, y \rangle$. Se x appartiene al dominio di f, allora esiste $\overline{y} \in \Sigma_1^*$ tale che $\overline{y} = f(x)$ e quindi $\langle x, \overline{y} \rangle \in L_f$. In questo caso, prima o poi, la stringa $\overline{y}$ verrà scritta sul secondo nastro e la computazione $T(x, \overline{y})$ terminerà in stato di accettazione e al passo 2.3 $T'(x)$ scriverà $\overline{y}$ sul nastro di output e terminerà. Questo dimostra che f è calcolabile. $\square$

Lezione 7 - Tesi di Church-Turing e Turing-equivalenza

Nelle lezioni precedenti abbiamo parlato di calcolabilità di funzioni e decidibilità di linguaggi sempre in relazione alle macchine di Turing. Potremmo ora domandarci se, utilizzando modelli di calcolo più potenti, non sia possibile calcolare funzioni non calcolabili dalle mdT.
Nella teoria della calcolabilità è stata proposta un'ampia gamma di modelli di calcolo, come ad esempio:

- grammatiche di tipo 0
- modello di Kleene basato sulle equazioni funzionali
- il λ-calcolo di Church
- ...

Tesi Church-Turing

Riguardo al potere computazionale di questi modelli, è stato dimostrato che sono tutti **Turing-equivalenti**, ovvero una funzione calcolabile con uno di quei modelli è anche calcolabile con una macchina di Turing.

Questa osservazione ha portato ad enunciare questa tesi, conosciuta come **tesi di Church-Turing**, che afferma che se la soluzione di un problema può essere descritta in una serie finita di *passi elementari*, allora esiste una macchina di Turing in grado di calcolarlo. In altri termini

> È calcolabile tutto (e solo) ciò che può essere calcolato da una macchina di Turing

La tesi afferma dunque che non esiste un modello di calcolo più potente della macchina di Turing.
Nel corso delle lezioni ci riferiremo a programmi scritti in un linguaggio specifico invece delle macchine di Turing. Questo linguaggio, oltre al concetto di variabili e collezioni di dati, utilizza le istruzioni seguenti:

- istruzione di assegnazione "$x \leftarrow y$"
- l'istruzione condizionale
  $\textbf{if } (\text{condizione}) \textbf{ then begin } \langle\text{istruzioni}\rangle \textbf{ end else begin } \langle\text{istruzioni}\rangle \textbf{ end}$ in cui la parte **else** può essere assente

- l'istruzione di loop $\textbf{while}\ (\text{condizione})\ \ \textbf{do begin}\ \langle\text{istruzioni}\rangle\ \textbf{end}$
- l'istruzione di output, che deve esssere l'ultima istruzione del programma che comunica il valore di una variabile $\textbf{Output:}\ \langle\text{nomeDiVariabile}\rangle$
  Se le sequenze di istruzioni sono istruzioni singole, si può omettere il $\textbf{begin end}$ e assumiamo che l'input venga comunicato al programma prima che le istruzioni inizino mediante l'istruzione $\textbf{Input:}$.
  Il linguaggio seguente lo denominiamo **PascalMinimo**.

Teorema 3.5

Per ogni programma scritto nel linguaggio **PascalMinimo** esiste una macchina di Turing T di tipo trasduttore che scrive sul nastro di output lo stesso valore fornito in output dal programma.

Idea della dimostrazione

Per la dimostrazione ci limitiamo al caso in cui il programma contenga solamente variabili semplici, ovvero non array. Sia $\mathscr{P}$ un programma scritto in PascalMinimo e che non contenga variabili strutturati. La macchina di Turing T è definita nel seguente modo:

- T, oltre ai nastri di input ed output, utilizza un nastro per ciascuna variabile e valore costante che compare in una condizione.
  - Ad esempio se in $\mathscr{P}$ compare l'istruzione $\textbf{if}\ (\text{a}=2)\ \textbf{then}\ \text{b}=1$, allora T utilizza un nastro per la variabile a, un nastro per la variabile b, un nastro per il valore costante 2 e un nastro per il valore costante 1. Infine T utilizza un nastro di lavoro per la valutazione delle espressioni e delle condizioni contenute nel programma.
- I contenuti del nastro sono codificati in binario
- Ad ogni istruzione di assegnazione in $\mathscr{P}$ corrisponde uno stato q_i con $i > 0$ in T
- Ad ogni istruzione condizionale in $\mathscr{P}$ corrisponde uno stato q_i oppure una coppia di stati $q_i, q_j : i, j > 0$, rispettivamente, nel caso in cui sia assente oppure presene la parte $\textbf{else}$
- Ad ogni istruzione di loop in $\mathscr{P}$ corrisponde uno stato $q_i : i > 0$ in T
- Lo stato iniziale di T è q_0

"FI/MOD II/img/img8.png" non trovato.

Vediamo quindi come costruire una quintupla partendo da un'istruzione in $\mathscr{P}$. Per semplicità, assumiamo di scrivere una singola istruzione per linea di codice, per avere una corrispondenza fra linee di codice e stati della macchina.

1. Ad ogni assegnazione di un valore ad una variabile, corrisponde una copia del nastro corrispondente alla costante o alla variabile sulla destra dell'assegnazione, sul nastro corrispondente alla variabile che deve prendere il valore (la parte sinistra dell'assegnazione). La sequenza termina con la macchina che entra nello stato corrispondente all'istruzione eseguita (ad esempio, prima di eseguire l'istruzione in linea 1, la macchina si trova nello stato q_0 e dopo averla eseguita entrerà in q_1).

2. Ad ogni assegnazione di una espressione ad una variabile (linea 7) corrisponde una sequenza di quintuple che eseguono quell'espressione sul nastro di lavoro e che terminano con la scrittura sul nastro corrispondente alla variabile che deve prendere il valore. La sequenza termina con la macchina che entra nello stato corrispondente all'istruzione eseguita.
3. Ogni condizione viene valutata, utilizzando il nastro di lavoro, confrontando i contenuti dei due nastri interessati. Ad esempio in linea 2 vengono confrontati i nastri corrispondenti alle variabili n e m. Dopo aver valutato la condizione, la macchina entra in uno stato che dipende dal valore della condizione e dal tipo di istruzione in cui è usata la condizione
   - In una istruzione **if-then-else** se la condizione è vera, la macchina entra in uno stato che permette di eseguire le istruzioni del ramo **if**, mentre se falsa, la macchina entra in uno stato che permette di eseguire le istruzioni del ramo **else**. Le quintuple seguenti indicano il comportamento delle linee 2-5

$$
\begin{array}{l}
\langle q_1, n > m, (\ldots, n, \ldots, m, \ldots), q_2, \cdot\rangle \\
\langle q_1, n \leq m, (\ldots, n, \ldots, m, \ldots), q_3, \cdot\rangle \\
\langle q_2, p \leftarrow n, (\ldots), q_4, \cdot\rangle \\
\langle q_3, p \leftarrow m, (\ldots), q_4, \cdot\rangle
\end{array}
$$

   - In una istruzione **while** se la condizione è vera allora la macchina entra in uno stato che permette di eseguire la prima istruzione del corpo del loop, altrimenti entra in uno stato che esegue la prima istruzione successiva al corpo del loop. Una volta eseguita l'ultima istruzione del corpo del loop, la macchina rientra nello stato di verifica della condizione del while. Le quintuple indicano il comportamento delle linee 6-8

$$
\begin{array}{l}
\langle q_4, p \geq 2, (\ldots, p, \ldots, 2, \ldots), q_5, \cdot\rangle \\
\langle q_4, p < 2, (\ldots, p, \ldots, 2, \ldots), q_6, \cdot\rangle \\
\langle q_5, (\ldots), p \leftarrow p - k, q_4, \cdot\rangle
\end{array}
$$

4. L'istruzione di output corrisponde alla scrittura sul nastro di output con la macchina che entra nello stato finale
5. Chiariamo ora come collegare le quintuple tra loro. Da quello scritto si intuisce che *lo stato con il quale la macchina termina una istruzione è lo stato che le consente di iniziare l'istruzione successiva*. Osserviamo che nelle quintuple descritte sopra, lo stato q_1 che corrisponde all'esecuzione della linea 1, è lo stato col quale inizia l'istruzione **if** alla linea 2. Questo è valido con solo due istruzioni:
   - lo stato col quale la macchina termina l'esecuzione di un blocco **if** deve passare il controllo all'istruzione *successiva all'* **else**
   - lo stato col quale la macchina termina l'esecuzione dell'ultima istruzione del corpo di un **while** deve passare il controllo all'istruzione *che testa la condizione del loop*

Lezione 8 - Equivalenza PascalMinimo mdT U

Abbiamo visto nella lezione precedente come il **PascalMinimo** non è un modello di calcolo più potente della macchina di Turing. In questa lezione dimostreremo la proposizione inversa, ovvero che le macchine di Turing non sono sistemi di calcolo più potenti del linguaggio **PascalMinimo**, dimostrando quindi la Turing-equivalenza.

Teorema 3.6
Per ogni macchina di Turing deterministica T di tipo riconoscitore ad un nastro esiste un programma $\mathscr{P}$ scritto in accordo alle regole del **PascalMinimo** tale che, per ogni stringa x, se $T(x)$ termina in uno stato finale $q_F \in \{q_A, q_R\}$ allora $\mathscr{P}$ con input x restituisce q_F.

Idea della dimostrazione
Siano $\langle q_{1_1}, s_{1_1}, s_{1_2}, q_{1_2}, m_1\rangle\langle q_{2_1}, s_{2_1}, s_{2_2}, q_{2_2}, m_2\rangle \dots \langle q_{k_1}, s_{k_1}, s_{k_2}, q_{k_2}, m_k\rangle$ le quintuple che descrivono la macchina di Turing T (dove q_{i_2} sarà lo stato di accettazione q_A e q_{j_2} sarà lo stato di rigetto q_R).
Nell'idea assumeremo che il movimento della testina di T sia rappresentato da un intero in $\{-1, 0, +1\}$ rispettivamente $\{s, f, d\}$

Possiamo quindi scrivere un programma $\mathscr{P}$ che si comporta come la macchina di Turing Universale. L'input del programma $\mathscr{P}$ è costituito, oltre che dalla parola x, anche dalla descrizione della macchina T che deve essere simulata, dalla descrizione di Σ, dall'insieme degli stati Q, dalle quintuple $\langle q_{1_1}, s_{1_1}, s_{1_2}, q_{1_2}, m_1\rangle \dots \langle q_{k_1}, s_{k_1}, s_{k_2}, q_{k_2}, m_k\rangle$ e dagli stati iniziali e finali di T.
Vengono utilizzate per il programma, le variabili seguenti:

- L'array $Q = \{q_1, q_2, \dots, q_m\}$ per gli stati
- L'array $\Sigma = \{s_1, s_2, \dots, s_n\}$ per l'alfabeto
- per le quinutple vengono usati gli array
  - $Q_1 = \{q_{1_1}, q_{2_1}, \dots, q_{k_1}\}$ che descrive gli stati di partenza di ciascuna quinutpla
  - $S_1 = \{s_{1_1}, s_{2_1}, \dots, s_{k_1}\}$ che descrive gli elementi di Σ che devono essere *letti* per poter eseguire ciascuna quintupla
  - $S_2 = \{s_{1_2}, s_{2_2}, \dots, s_{k_2}\}$ che descrive gli elementi di Σ che devono essere *scritti* per poter eseguire ciascuna quintupla
  - $Q_2 = \{q_{1_2}, q_{2_2}, \dots, q_{k_2}\}$ che descrive gli stati di arrivo di ciascuna quintupla
  - $M = \{m_1, m_2, \dots, m_k\}$ che descrive i movimenti della testina che avvengono quando è eseguita ciascuna quintupla

In questo modo la quintupla $\langle q_{j_1}, s_{j_1}, s_{j_2}, q_{j_2}, m_j \rangle$ è rappresentata da $Q_1[j], S_1[j], S_2[j], Q_2[j], M[j]$

- lo stato finale q_0 e i due stati finali q_A e q_R
  Nel programma $\mathscr{P}$ vengono anche utilizzate le seguenti variabili
- q, che descrive lo stato attuale della macchina
- N, un array che descrive il contenuto del nastro ad ogni istante della computazione; la dimensione di N, ad ogni istante, è pari alla porzione di nastro utilizzato dalla macchina T
- t la posizione della testina sul nastro di T
- primaCella, che ad ogni istante memorizza l'indirizzo della cella del nastro più a sinistra utilizzata fino ad allora dalla computazione $T(x)$
- ultimaCella, che ad ogni istante memorizza l'indirizzo della cella del nastro più a destra utilizzata fino ad allora dalla computazione $T(x)$
- i e j variabili di iterazione
- $\texttt{trovata}$ è una variabile booleana utilizzata per verificare il ritrovamento della corretta quintupla da eseguire

“FI/MOD II/img/img9.png” non trovato.

In figura viene descritto il programma $\mathscr{P}$ che simula la macchina di Turing universale.
Esso consiste in un unico loop tra le linee 5-25 nel quale, fissato lo stato attuale q e la posizione di t sul nastro e quindi il simbolo letto $N[t]$, cerca la quintupla che inizia con la coppia $(q, N[t])$ e se esiste, la esegue. Le variabili primaCella e ultimaCella vengono utilizzate nel caso in cui il movimento della testina vada oltre alla lunghezza della stringa memorizzata in N, in quel caso ne aggiorna la lunghezza.

Lezione 9 - Halting Problem

Cantor ha dimostrato che esistono insiemi infiniti "piccoli" e "grandi", basandosi sul concetto di corrispondenza biunivoca, definendo la "grandezza" di un insieme infinto con il termine *numero transfinito*. Ha dimostrato anche che non esiste una corrispondenza biunivoca fra l'insieme dei numeri naturali e l'insieme dei numeri reali, provando che l'insieme dei numeri reali $\mathbb{R}$ è strettamente più grande dell'insieme dei numeri reali $\mathbb{N}$.

In questa lezione siamo pronti a rispondere alla seguente domanda

> *Esiste un problema che non può essere risolto?*

Nei paragrafi 5.1 e 5.2 della dispensa 5 viene dimostrato che **esiste un problema irrisolvibile** secondo le seguenti osservazioni: siccome le macchine di Turing sono tante quanti i numeri naturali e, poiché i problemi sono tanti quanti i numeri reali, allora esiste almeno un problema che non corrisponde a nessuna macchina di Turing e quindi **non può essere risolto**.
Questo però non ci dà l'idea di come possa essere fatto un problebma irrisolvibile, ma Turing ha costruito un problema irrisolvibile.

Nella lezione 5 avevamo descritto una macchina di Turing T con alfabeto $\Sigma = \{0, 1\}$, l'insieme degli stati $Q_T = \{\omega_0, \ldots, \omega_{k-1}\}$ con stato iniziale ω_0, stato di accettazione ω_1 e stato di rigetto ω_2 e l'insieme delle quintuple $P = \{p_1, \ldots, p_n\}$ dove la i-esima quitupla è

$$p_i = \langle \omega_{i_1}, b_{i_1}, b_{i_2}, \omega_{i_2}, m_i \rangle$$

tramite la parolona

$$\rho_T = \omega_0 - \omega_1 \otimes \omega_{1_1} - b_{1_1} - b_{1_2} - \omega_{1_2} - m_1 \oplus \cdots \oplus \omega_{h_1} - b_{h_1} - b_{h_2} - \omega_{h_2} - m_h \oplus$$

Poi, avevamo introdotto la codifica binaria b^Q dell'insieme degli stati Q_T degli stati di T nel seguente modo:

- $b^Q = Q_T \rightarrow \{0, 1\}^k$, ovvero la codifica di uno stato in una parola di k bit
- $b^Q(\omega_i)$ è la parola che ha un 1 in posizione $i + 1$ e 0 altrove, esempio $b^Q(\omega_0) = 1000$
  e quindi rappresentato la nostra macchina T con la parola β_T nell'alfabeto

$\Sigma = \{0, 1, \oplus, \otimes, -, f, s, d\}$ nel seguente modo

$$\beta_T = b^Q(w_0) - b^Q(w_1) \otimes b^Q(w_{1_1}) - b_{1_1} - b_{1_2} - b^Q(\omega_{1_2}) - m_1 \oplus \cdots \oplus$$

Quello che viene mostrato nel paragrafo 5.1 è trasformare la parola β_T in un numero sostituendo in questo modo:

- $s \to 5$
- $f \to 6$
- $d \to 7$
- $- \to 4$
- $\oplus \to 3$
- $\otimes \to 2$
- $\square \to 9$

dopo le sostituzioni premettiamo il numero 2 alla parola ottenuta. In questo modo abbiamo associato ad ogni macchina di Turing un numero intero.

> **Oss.**
> L'associazione è univoca, infatti a macchine di Turing diverse sono associati interni diversi

Halting Problem

Turing considerò il seguente linguaggio, sottoinsieme di $\mathbb{N} \times \mathbb{N}$

$$L_H = \{(i, x) \in \mathbb{N} \times \mathbb{N} : i \text{ è la codifica di una macchina di Turing } T_i \text{ e } T_i(x) \text{ termina}\}$$

che è detto **Halting Problem** e Turing dimostrò che L_H è accettabile e non decidibile (quindi L_H^C non è accettabile). Questo lo dimostreremo tra poco.

Ora cerchiamo di capire che senso ha domandarsi se dato $(i, x) \in \mathbb{N} \times \mathbb{N}$ allora $(i, x) \in L_H$ e lo facciamo con un piccolo esempio.

> **Esempio**
> Supponiamo di aver scritto un programma ed averlo lanciato su un input x, che è un istanza del problema risolto dal nostro programma. Attendiamo per molto tempo la risposta, finché sorge un dubbio: **e se fosse andato in loop?**
> Supponiamo allora che esista un programma che, preso in input un programma P e un suo input x, mi dice se l'esecuzione di P su x termina oppure no. Questo programma decide l'**Halting Problem**!

Teorema 5.4

L_H è un linguaggio accettabile

Dim.

Dobbiamo mostrare che esiste una macchina di Turing T che per ogni $(i, x) \in \mathbb{N} \times \mathbb{N}$ allora

$$O_T(i, x) = q_A \iff (i, x) \in L_H$$

La macchina che cerchiamo è una modifica della macchina universale U, che chiamiamo U', quindi una macchina a 4 nastri. Su N_1 scriviamo il numero i in notazione decimale e su N_2 scriviamo $x \in \{0, 1\}^*$. U' inizia la sua computazione verificando che i non contenga cifre 8 e 9 e che inizi con la cifra 2: se non è così la macchina rigetta, altrimenti cancella il 2 iniziale e traduce quello che rimane nell'alfabeto di lavoro Σ di U.
In seguito U' simula la computazione di U e se U termina (sia nello stato di accettazione che di rigetto) allora U' termina nello stato di accettazione

Oss.

Se i non è la codifica di una macchina di Turing, allora, poiché l'insieme delle quintuple di una qualsiasi macchina di Turing è totale e le computazioni con un input che non *rispettano le specifiche* non terminano, allora $U'(i, x)$ non termina.

Sia $(i, x) \in L_H$: allora, la computazione $T_i(x)$ termina, e quindi, la computazione $U'(i, x)$ accetta.
Viceversa, sia $(i, x) \in \mathbb{N} \times \mathbb{N}$ tale che $U'(i, x)$ accetta; poiché la computazione $U'(i, x)$ simula la computazione $U(i, x)$, allora anche $U(i, x)$ termina e, dunque, i è la codifica di una macchina di Turing e $T_i(x)$ termina, quindi $(i, x) \in L_H$. $\square$

Teorema 5.5

Il linguaggio L_H non è decidibile

Per la dimostrazione andiamo prima a modificare la notazione. Osserviamo che, data una macchina T ed un suo input x, il valore $O_T(x)$ è definito solo per gli x tali che la computazione $T(x)$ termina. Poiché nella dimostrazione dobbiamo usare spesso la possibilità che una computazione non termini, indicheremo con $T(x)$ sia la computazione dalla macchina T sull'input x che il suo esito. Assumeremo quindi

$$T(x) = \begin{cases} q \in Q_f & \text{se la computazione } T(x) \text{ termina} \\ \\ \text{non termina} & \text{se la computazione } T(x) \text{ non termina} \end{cases}$$

Dim.

Procederemo per assurdo. Supponiamo che L_H sia decidibile. Allora deve esistere una macchina di Turing T tale che

$$T(i,x) = \begin{cases} q_A & \text{se } (i,x) \in L_H \\ \\ q_R & \text{se } (i,x) \notin L_H \end{cases}$$

Assumiamo che T sia una macchina ad un nastro.
Da T possiamo derivare una nuova macchina $T^{'}$, complementando gli stati di accettazione e rigetto di T, che, terminando su ogni input (come T), accetta tutte e sole le coppie $(i,x) \in \mathbb{N} \times \mathbb{N} - L_H$, ossia

$$T^{'}(i,x) = \begin{cases} q_R & \text{se } (i,x) \in L_H \\ \\ q_A & \text{se } (i,x) \notin L_H \end{cases}$$

A partire da $T^{'}$ deriviamo una terza macchina $T^{''}$, che accetta (i,x) se $(i,x) \notin L_H$ mentre **non termina** se $(i,x) \in L_H$. Con la coppia (i,x) sul nastro, $T^{''}$ invoca $T^{'}$ passandogli (i,x) come parametri, risulta quindi

$$T^{''}(i,x) = \begin{cases} q_A & \text{se } T^{'}(i,x) \text{ termina in } q_A \\ \\ \text{entra in loop} & \text{se } T^{'}(i,x) \text{ termina in } q_R \end{cases}$$

per questo è sufficiente aggiungere le quintuple $\langle q_R, y, y, q_R, f \rangle$ per ogni $y \in \{0,1\}$ e rimuovendo q_R dagli stati finali di $T^{''}$

Oss.
Poiché $(i,x) \in \mathbb{N} \times \mathbb{N}$ è una coppia di interi, allora (i,i) può essere dato in pasto alle tre macchine $T, T^{'}, T^{''}$. Se i è la codifica di una macchina di Turing, allora:

- $T(i,i)$ accetta se $(i,i) \in L_H$, ossia se $T_i(i)$ termina e rigetta se $(i,i) \notin L_H$ ossia se $T_i(i)$ non termina
- $T^{'}(i,i)$ accetta se $(i,i) \notin L_H$ ossia se $T_i(i)$ non termina e $T^{'}(i,i)$ rigetta se $(i,i) \in L_H$, ossia se $T_i(i)$ termina
- $T^{''}(i,i)$ accetta se $(i,i) \notin L_H$ ossia se $T_i(i)$ non termina e $T^{''}(i,i)$ *non termina* se $(i,i) \in L_H$, ossia se $T_i(i)$ termina

A partire da $T^{''}$ costruiamo l'ultima macchina T^* che *lavora con un solo input* e tale che **l'esito della computazione $T^*(i)$ coincide con la computazione di $T^{''}(i,i)$**, quindi se i è la codifica di una macchina di Turing allora

$$T^*(i) = \begin{cases} q_A & \text{se } (i,i) \notin L_H \\ \text{non termina} & \text{se } (i,i) \in L_H \end{cases}$$

Siccome abbiamo supposto che T esiste, allora anche T^* esiste e la possiamo codificare con un intero secondo le sostituzioni viste prima. Indichiamo con k il codice di T^* ottenuto applicando le sostituzioni, quindi $T^* = T_k$ e siccome k è un numero intero, può essere input di T^* e quindi di T_k. Ci chiediamo ora *qual è l'esito della computazione $T_k(k)$?*

- Se $T_k(k) = T^*(k)$ accettasse, allora $T^{'}(k,k)$ dovrebbe accettare anch'essa. Ma se $T^{'}(k,k)$ accetta, allora, poiché k è la codifica di una macchina di Turing, allora $(k,k) \notin L_H$ e quindi $T_k(k)$ non termina. Dunque **$T_k(k) = T^*(k)$ accetta solo se $T_k(k) = T^*(k)$ non termina**.
- $T_k(k) = T^*(k)$ non termina solo se $(k,k) \in L_H$, ossia se $T_k(k)$ termina, quindi **$T_k(k) = T^*(k)$ non termina solo se $T_k(k) = T^*(k)$ termina**.
  Quindi entrambi le ipotesi portano ad una contraddizione e poiché abbiamo supposto che L_H è decidibile, abbiamo costruito una macchina che decide il linguaggio che in realtà non può esistere, portandoci alla conclusione che abbiamo sbagliato la supposizione che **L_H sia decidibile**, dimostrando invece il contrario. $\square$

Cosa significa quindi che l'Halting Problem è accettabile ma non decidibile? Ricordiamoci che *un linguaggio L è decidibile se, L è accettabile ed L^C è accettabile* e allora poiché L_H è accettabile ma non decidibile, concludiamo dicendo che L_H^C non è accettabile.

Lezione 10 - Riduzioni

Nella dimostrazione dell'indecidibiltà dell'Halting Problem siamo partiti supponendo di avere una macchina T in grado di decidere L_H e, *senza sapere come era fatta la macchina T*, abbiamo costruito altre macchine $T^{'}, T^{''}$ e T^* che ci hanno portato a dimostrare la non decidibilità. Questo utilizzo "a scatola nera" corrisponde al concetto di invocazione di funzione, come nella programmazione. Quando $T^{'}$ usava T, le passava "come parametro" il suo input (i, x).
Generalmente potremmo utilizzare una macchina T_0 all'interno di una macchina T_1, perché:

- il linguaggio deciso/accettato da T_0 potrebbe essere anche diverso da quello deciso/accettato da T_1
- potrebbe essere necessario "modificare" l'input di T_1 prima di darlo in pasto a T_0

> **Esempio**
> Voglio costruire una macchina che decida il linguaggio L_{P12} che contiene le parole palindrome di lunghezza pari sull'alfabeto $\{1, 2\}$. Notiamo che, in effetti, questo linguaggio somiglia molto al linguaggio L_{PPAL} di tutte le stringhe palindrome di lunghezza pari sull'alfabeto $\{a, b\}$. Quindi, piuttosto che riprogettare una macchina, potremmo utilizzare T_{PPAL}, l'unico problema da risolvere riguarda l'alfabeto su cui la macchina deve lavorare. L'idea potrebbe essere quella di trasformare le parole di L_{P12} in parole di L_{PPAL}.
> Prendo quindi il mio $x \in \{1, 2\}^*$ e assumendo $x = x_1 x_2 \dots x_n$, per ogni $h = 1, 2, \dots, n$ procedo così:
>
> - se $x_h = 1$ allora poniamo $y_h = a$
> - se $x_h = 2$ allora poniamo $y_h = b$
>   Ho qundi costruito 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 $y \notin L_{PPAL}$

Quello che abbiamo fatto nell'esempio, può essere vista come la progettazione di una funzione $f : \{1, 2\}^* \rightarrow \{a, b\}^*$ tale che:

- f è calcolabile e totale, ossia
  - è definita per ogni $x \in \{1, 2\}^*$
  - esiste una macchina di Turing T_f di tipo trasduttore tale che, per ogni parola $x \in \{1, 2\}^*$, la computazione $T_f(x)$ termina con la parola $f(x) \in \{a, b\}^*$ scritta sul nastro di output

- per ogni $x \in \{1,2\}^*$ vale che $x \in L_{P12}$ se e solo se $f(x) \in L_{PPAL}$
  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 $L_{P12} \preceq L_{PPAL}$.

Riduzioni

Generalizziamo quello che abbiamo detto ora.
Def. Riduzione many-to-one
Dati due linguaggi $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$, diciamo che **L_1 è riducibile ad L_2** e scriviamo $L_1 \preceq L_2$ se esiste una funzione $f : \Sigma_1^* \to \Sigma_2^*$ tale che:

- f è totale e calcolabile, ossia
  - è definita per ogni $x \in \Sigma_1^*$
  - esiste una macchina di Turing T_f di tipo trasduttore 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
- per ogni $x \in \Sigma_1^*$ vale che $x \in L_1 \iff f(x) \in L_2$

Il concetto di riducibilità funzionale ci permette di correlare tra loro linguaggi in modo tale che:

- la decidibilità/accettabilità di un linguaggio implichi la decidibilità/accettabilità dei linguaggi ad esso riducibili
- la non decidibilità/accettabilità di un linguaggio implichi la non decidibilità/accettabilità dei linguaggi cui esso si riduce

Se dimostro che, dato un linguaggio L_3, vale $L_3 \preceq L_4$ per un altro linguaggio L_4, se io so che L_4 è decidibile, allora posso concludere che L_3 *è decidibile*.
Infatti, siano $L_3 \subseteq \Sigma_3^*$ e $L_4 \subseteq \Sigma_4^*$:

- **L_4 è decidibile**, allora esiste una macchina T_4 tale che, per ogni $x \in \Sigma_4^*$, $T(x)$ termina in q_A se $x \in L_4$ e termina in q_R se $x \notin L_4$
- $L_3 \preceq L_4$, allora esiste 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 $y = f(x)$ sul secondo nastro
- poi simula $T_4(y)$ che, se accetta allora anche T_3 accetta e se rigetta allora anche T_3 rigetta
  Abbiamo quindi dimostrato che L_3 è decidible, applicando la riduzione.

Possiamo utilizzare la riduzione per dimostrare **la non decidibilità/accettabilità** di un linguaggio. Dato un linguaggio L_2, se dimostro che $L_1 \preceq L_2$ e 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 $L_1 \preceq L_2$, per quanto visto nelle righe sopra, anche L_1 dovrebbe essere decidibile, contraddicendo l'ipotesi per cui L_1 è non decidibile.

Riducendo un linguaggio "sconosciuto" ad un linguaggio decidibile/accettabile, dimostriamo che il linguaggio "sconosciuto" è anch'esso accettabile/decidibile

- dimostrando che $L_{P12} \preceq L_{PPAL}$ e sapendo che L_{PPAL} è **decidibile**, abbiamo mostrato che L_{P12} è decidibile
- dimostrando che $L_{H0} \preceq L_H$ e sapendo che L_H è *accettabile*, abbiamo dimostrato che L_{H0} è accettabile

Riducendo un linguaggio decidibile/accettabile ad un linguaggio "sconosciuto", dimostriamo che il linguaggio "sconosciuto" è anch'esso non accettabile/decidibile

- dimostrando che $L_H \preceq L_{H0}$ e sapendo che L_H è non decidibile, abbiamo dimostrato che L_{H0} è non decidibile

> **Esempio**
> Consideriamo il linguaggio
>
> $$L_{H0} = \{i \in \mathbb{N} : \ i \text{ è la codifica di una macchina di Turing e } T_i(0) \text{ termina}\}$$
>
> ovvero, $i \in L_{H0}$ se i è la codifica di una macchina di Turing T_i e la computazione di T_i, con input la parola costituita dal carattere 0, termina.
> Osserviamo quanto questo problema somiglia molto all'Halting problem, infatti L_{H0} è un sottoinsieme di L_H che contiene solo le coppie $(i, 0)$ e questo si esprime dicendo che *L_{H0} è una restrizione di L_H*, quindi una macchina che si occupa di L_H può occuparsi di L_{H0} e quindi la macchina che accetta L_H accetterà anche L_{H0}.
> Osserviamo anche che, però, L_{H0} sembra più facile di L_H, quindi magari è decidibile... ma questo in realtà non è vero e lo dimostriamo **riducendo L_H ad L_{H0}**, ossia individuando una funzione totale e calcolabile f che trasforma tutte le parole di L_H
> in parole di L_{H0} e parole non in L_H in parole non in L_{H0}

Consideriamo una coppia $(i, x) \in \mathbb{N} \times \mathbb{N}$ con $x = x_1 x_2 \dots x_n$ e da essa deriviamo un intero $k = k_{(i,x)} = f(i, x)$ nel modo seguente:

- 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
- se i *è la codifica di una macchina di Turing*, allora costruiamo una macchina di Turing $M_{(i,x)}$ che, con input 0, simula $T_i(x)$ nel seguente modo

1. nello stato iniziale q_1 se legge 0 scrive x_1 ed entra in q_2 muovendosi a destra, altrimenti rigetta
2. nello stato q_2 se legge $\square$ sul nastro scrive x_2 ed entra in q_3 muovendosi a destra, altrimenti rigetta
3. $\vdots$
4. nello stato q_n se legge $\square$ sul nastro scrive x_n, riposiziona la testina sul primo carattere in input ed entra nello stato iniziale q_0 di T_i, altrimenti rigetta

Oss. 1
Il numero degli stati $M_{(i,x)}$ dipende da x e potrebbe sembrare non costante, ma in realtà non è così perché:

- x non è input per $M_{(i,x)}$ che si attende come input il solo carattere 0
- ricordiamo che abbiamo considerato una coppia (i, x) e **solo dopo averla fissata** abbiamo costruito $M_{(i.x)}$

$k_{(i,x)}$ corrisponde alla codifica di $M_{(i,x)}$ e per questo $T_{k_{(i,x)}} = M_{(i,x)}$ allora $k_{(i,x)} \in L_{H0}$ se e solo se $T_{k_{(i,x)}}(0) = M_{(i,x)}(0)$ termina e resta da capire in quali casi termina.
Per ogni $(i, x) \in \mathbb{N} \times \mathbb{N}$, se $M_{(i,x)}(0)$ termina, allora i è la codifica di una macchina di Turing e $T_i(x)$ termina. Quindi, se $k_{(i,x)} \in L_{H0}$ allora $T_{k_{(i,x)}}(0) = M_{(i,x)}(0)$ termina e allora i è la codifica di una macchina di Turing e $T_i(x)$ termina.
In generale se $k_{(i,x)} \in L_{H0}$ allora $(i, x) \in L_H$.

Per ogni $(i, x) \in \mathbb{N} \times \mathbb{N}$, se **non** $M_{(i,x)}(0)$ termina, allora ci sono due casi:

- $M_{(i,x)}$ è stata costruita secondo il primo caso (loop) e quindi i non è la codifica di una macchina di Turing T_i
- $M_{(i,x)}$ è stata costruita secondo il secondo caso e i è la codifica di una macchina di Turing, ma $T_i(x)$ **non termina**.
  Ricapitolando se $k_{(i,x)} \notin L_{H0}$ allora $T_{k_{(i,x)}}(0) = M_{(i,x)}(0)$ non termina e allora $(i, x) \notin L_H$.
  In generale se $k_{(i,x)} \notin L_{H0}$ allora $(i, x) \notin L_H$.

Abbiamo calcolato $k_{(i,x)}$ come la codifica di $M_{(i,x)}$ ponendo $k_{(i,x)} = f(i, x)$ e siccome abbiamo descritto il procedimento per calcolare $k_{(i,x)}$ per ogni coppia (i, x), allora f è totale e calcolabile per ogni $(i, x) \in \mathbb{N} \times \mathbb{N}$ vale che $(i, x) \in L_H$ se e solo se $k_{(i,x)} = f(i, x) \in L_{H0}$.

Quindi $L_H \preceq L_{H0}$ e questo dimostra che L_{H0} non è decidibile.

Lezione 12 - Classi di complessità

In questa lezione iniziamo il processo di classificazione dei linguaggi in base alle risorse tempo e spazio sufficienti alla loro decisione/accettazione.
Vediamo intanto due teoremi, che mostrano che il numero di istruzioni e di celle che occorrono per *decidere deterministicamente* un linguaggio possono essere individuati solo *a meno di costanti moltiplicative*.

Teorema 6.6 (Compressione lineare)
Sia $L \subseteq \Sigma^*$ un linguaggio deciso da una macchina di Turing deterministica ad un nastro T tale che, per ogni $x \in \Sigma^*$, $\text{dspace}(T, x) = s(|x|)$ e sia $k > 0$ una costante. Allora esiste una macchina di Turing ad un nastro $T^{'}$ che decide L e per ogni $x \in \Sigma^*$
$\text{dspace}(T^{'}, x) \leq \frac{s(|x|)}{c} + O(|x|)$.

Questo teorema ci dice che, dato un qualunque algoritmo, esiste sempre un algoritmo che è una frazione costante della memoria del primo. La presenza dell'addendo $O(|x|)$ deriva dal fatto che l'input di $T^{'}$ è lo stesso di T, quindi $T^{'}$ deve per prima cosa codificare in forma compressa il proprio input e poi lavorare sull'alfabeto compresso.

Oss.
L'alfabeto compresso è Σ^k (ovvero un carattere dell'alfabeto compresso è una parola di k caratteri di Σ) e l'alfabeto di $T^{'}$ è $\Sigma^k \cup \Sigma$.

Teorema 6.7 (Accelerazione lineare)
Sia $L \subseteq \Sigma^*$ un linguaggio deciso da una macchina di Turing deterministica ad un nastro T tale che, per ogni $x \in \Sigma^*$, $\text{dtime}(T, x) = t(|x|)$ e sia $k > 0$ una costante. Allora:

- esiste una macchina di Turing *ad un nastro* $T^{'}$ che decide L e, per ogni $x \in \Sigma^*$
  $\text{dtime}(T^{'}, x) \leq \frac{t(|x|)}{k} + O(|x|^2)$
- esiste una macchina di Turing *a due nastri* $T^{''}$ che decide L e, per ogni $x \in \Sigma^*$
  $\text{dtime}(T^{''}, x) \leq \frac{t(|x|)}{k} + O(|x|)$

Questo teorema ci dice che, dato un qualunque algoritmo, esiste sempre un algoritmo più veloce del primo di un fattore costante.
La presenza degli addendi $O(|x|^2)$ e $O(|x|)$ deriva dal fatto che, per poter essere più veloci, le macchine $T^{'}$ e $T^{''}$ devono innanzitutto codificare in forma compressa il proprio input: se la codifica

compressa viene scritta su un nastro apposito (come fa T'' sul secondo nastro) sono sufficienti $O(|x|)$ passi, se invece si dispone di un solo nastro occorrono $O(|x|^2)$ passi.

Classi di complessità

Possiamo raggruppare i linguaggi in base all'efficienza delle macchine che li decidono, ad esempio potremmo considerare l'insieme dei linguaggi tali che, **la miglior macchina che li decide**, ha una certa efficienza.
Vediamo cosa vuol dire. Un linguaggio L è un insieme di parole, ed una macchina che decide L esegue un numero diverso di operazioni quando opera su input diversi. Pensiamo ad esempio alla macchina T_{PPAL} con due input diversi. La computazione $T_{PPAL}(abababab)$ rigetta dopo aver eseguito 10 quintuple, mentre la computazione $T_{PPAL}(abbbbbbba)$ accetta dopo aver eseguito circa 45 quintuple.
Ma cosa significa che una macchina che decide un linguaggio ha una certa efficienza? La risposta è che vogliamo che la macchina che decide un linguaggio $L \subseteq \Sigma^*$ si comporti "bene" su ogni parola $x \in \Sigma^*$. Osserviamo che non possiamo scegliere la migliore macchina che decide un linguaggio, perché se il linguaggio fosse deciso da una macchina con una certa efficienza, lo stesso linguaggio è deciso da una macchina efficiente il doppio, il triplo, il quadruplo e così via.

Per risolvere questa questione ricorriamo alla notazione $O(f(x))$: diciamo che *un linguaggio L appartiene all'insieme caratterizzato dall'efficienza temporale individuata dalla funzione totale e calcolabile f se* **esiste** *una macchina T che decide (o accetta) L e che, per ogni parola x sull'alfabeto di L, termina in $O(f(|x|))$ istruzioni.

Le classi che misurano "efficienza temporale" nel caso deterministico si chiamano DTIME
Data una funzione totale e calcolabile f

$$\begin{aligned}\mathrm{DTIME}[f(n)] = \{L \subseteq \{0,1\}^* & \text{ tali che esiste una macchina deterministica T che decide L e,} \\ & \text{per ogni } x \in \{0,1\}^*, \mathrm{dtime}(T,x) \in O(f(|x|))\}\end{aligned}$$

Le classi che misurano "efficienza spaziale" nel caso deterministico si chiamano DSPACE
Data una funzione totale e calcolabile f

$$\begin{aligned}\mathrm{DSPACE}[f(n)] = \{L \subseteq \{0,1\}^* & \text{ tali che esiste una macchina deterministica T che decide L} \\ & \text{e per ogni } x \in \{0,1\}^*, \mathrm{dspace}(T,x) \in O(f(|x|))\}\end{aligned}$$

Possiamo anche definire le classi di complessità nel caso non deterministico.

Le classi che misurano "efficienza temporale" nel caso non deterministico si chiamano NTIME.
Data una funzione totale e calcolabile f

$$\mathrm{NTIME}[f(n)] = \{L \subseteq \{0,1\}^* \text{ tali che esiste una macchina non deterministica NT} \\ \text{che accetta L e per ogni } x \in L, \mathrm{ntime}(NT, x) \in O(f(|x|))\}$$

Oss.

Perché una classe non deterministica è definita in base al tempo di accettazione, invece che di decisione? Se sappiamo che un linguaggio è accettato in un certo numero di istruzioni, sappiamo che quel linguaggio è decidibile, ma non sappiamo quanto tempo occorre a rigettare le parole del suo complemento, ma quello che interessa a noi è accettare le parole del linguaggio.

Le classi che misurano "efficienza spaziale" nel caso non deterministico si chiamano NSPACE.
Data una funzione totale e calcolabile f

$$\mathrm{NSPACE}[f(n)] = \{L \subseteq \{0,1\}^* \text{ tali che esiste una macchina non deterministica NT} \\ \text{che accetta L e per ogni } x \in L, \mathrm{nspace}(NT, x) \in O(f(|x|))\}$$

Classi complemento

Sia f una funzione totale e calcolabile.
La classe $\mathrm{coDTIME}[f(n)]$ contiene i linguaggi il cui complemento è contenuto in $\mathrm{DTIME}[f(n)]$

$$\mathrm{coDTIME}[f(n)] = \{L \subseteq \{0,1\}^* \text{ tali che } L^c \in \mathrm{DTIME}[f(n)]\}$$

La classe $\mathrm{coDSPACE}[f(n)]$ contiene i linguaggi il cui complemento è contenuto in $\mathrm{DSPACE}[f(n)]$

$$\mathrm{coDSPACE}[f(n)] = \{L \subseteq \{0,1\}^* \text{ tali che } L^c \in \mathrm{DSPACE}[f(n)]\}$$

La classe $\mathrm{coNTIME}[f(n)]$ contiene i linguaggi il cui complemento è contenuto in $\mathrm{NTIME}[f(n)]$

$$\mathrm{coNTIME}[f(n)] = \{L \subseteq \{0,1\}^* \text{ tali che } L^c \in \mathrm{NTIME}[f(n)]\}$$

La classe $\mathrm{coNSPACE}[f(n)]$ contiene i linguaggi il cui complemento è contenuto in $\mathrm{NSPACE}[f(n)]$

$$\mathrm{coNSPACE}[f(n)] = \{L \subseteq \{0,1\}^* \text{ tali che } L^c \in \mathrm{NSPACE}[f(n)]\}$$

Oss.

Consideriamo i linguaggi definiti sull'alfabeto $\{0, 1\}$ per comodità, ma potremmo utilizzare un alfabeto qualsiasi, infatti se un linguaggio è deciso da una macchian definita su un alfabeto qualsiasi, allora esiste anche una macchina definita su $\{0, 1\}$ che lo decide.

Alla funzione f che definisce una classe di complessità (ad esempio $\text{coDTIME}[f(n)]$), diamo il nome di **funzione limite**.

Relazioni fra le classi di complessità

Teorema 6.8
Per ogni funzione calcolabile e totale $f : \mathbb{N} \to \mathbb{N}$

$$\text{DTIME}[f(n)] \subseteq \text{NTIME}[f(n)] \; e \; \text{DSPACE}[f(n)] \subseteq \text{NSPACE}[f(n)]$$

Dim.
Basta osservare che una macchina di Turing deterministica è una particolare macchina di Turing non deterministica con grado di non determinismo pari ad 1 e che ogni parola decisa in k passi è anche accettata in k passi. $\square$

Teorema 6.9
Per ogni funzione calcolabile e totale $f : \mathbb{N} \to \mathbb{N}$

$$\text{DTIME}[f(n)] \subseteq \text{DSPACE}[f(n)] \; e \; \text{NTIME}[f(n)] \subseteq \text{NSPACE}[f(n)]$$

Dim. (analogo il caso non deterministco)
Segue dal Teorema 6.1. Sia $L \subseteq \{0, 1\}^*$ tale che $L \in \text{DTIME}[f(n)]$: allora esiste una macchina di Turing deterministica T che decide L e tale che, per ogni $x \in \{0, 1\}^*$.
$\text{dtime}(T, x) \in O(f(|x|))$. Poiché $\text{dspace}(T, x) \leq \text{dtime}(T, x)$, allora
$\text{dspace}(T, x) \leq \text{dtime}(T, x) \in O(f(|x|))$. Questo implica che $\text{dspace}(T, x) \in O(f(|x|))$ e dunque $L \in \text{DSPACE}[f(n)]$. $\square$

Teorema 6.10
Per ogni funzione calcolabile e totale $f : \mathbb{N} \to \mathbb{N}$

$$\text{DSPACE}[f(n)] \subseteq \text{DTIME}[2^{O(1)f(n)}] \;\; e \;\; \text{NSPACE}[f(n)] \subseteq \text{NTIME}[2^{O(1)f(n)}]$$

Dim.
Segue dal Teorema 6.1. Sia $L \subseteq \{0, 1\}^*$ tale che $L \in \text{DSPACE}[f(n)]$: allora esiste una macchina di Turing deterministica T che decide L e tale che, per ogni $x \in \{0, 1\}^*$,
$\text{dspace}(T, x) \in O(f(|x|))$.

Poiché

$$\text{dtime}(T, x) \leq \text{dspace}(T, x)|Q|(|\Sigma| + 1)^{\text{dspace}(T,x)} = \text{dspace}(T, x)|Q|3^{\text{dspace(T,x)}} =$$

$$= 2^{\log(\text{dspace}(T,x))}|Q|\left[2^{\log(3)}\right]^{\text{dspace(T,x)}}$$

$$= |Q|2^{\log(\text{dspace}(T,x))+\text{dspace}(T,x)\log(3)} \leq |Q|2^{[1+\log(3)]\ \text{dspace}(T,x)}$$

allora $\text{dtime}(T, x) \in O(2^{O(1)f(|x|)})$ e dunque $L \in \text{DTIME}[2^{O(1)f(n)}]$. □

Teorema 6.11

Per ogni funzione calcolabile e totale $f : \mathbb{N} \to \mathbb{N}$

$$\text{DTIME}[f(n)] = \text{coDTIME}[f(n)] \ \ e \ \ \text{DSPACE}[f(n)] = \text{coDTIME}[f(n)]$$

Dim.

Sia $L \subseteq \{0, 1\}^*$ tale che $L \in \text{DTIME}[f(n)]$: allora, esiste una macchina di Turing deterministica T che decide L e tale che, per ogni $x \in \{0, 1\}^*$, $\text{dtime}(T, x) \in O(f(|x|))$.
Poiché T decide L, allora $T(x) = q_A$ se $x \in L$ e $T(x) = q_R$ se $x \in \{0, 1\}^* - L = L^C$.
Costruiamo una macchina T', identica a T, ma con gli stati di accettazione e rigetto invertiti.
Allora e $T'(x) = q_R$ se $x \in L$ e $T(x) = q_A$ se $x \in \{0, 1\}^* - L = L^C$. Quindi T' decide L^C e, per ogni $x \in \{0, 1\}^*$, $\text{dtime}(T, x) \in O(f(|x|))$, quindi $L^C \in \text{DTIME}[f(n)]$.
Poiché L è un qualunque linguaggio in $\text{DTIME}[f(n)]$ e, quindi, L^C è un qualunque linguaggio in $\text{coDTIME}[f(n)]$, questo significa che:

- per ogni linguaggio $L^C \in \text{coDTIME}[f(n)]$, $L^C \in \text{DTIME}[f(n)]$ - ovvero $\text{coDTIME}[f(n)] \subseteq \text{DTIME}[f(n)]$
- per ogni linguaggio $L \in \text{DTIME}[f(n)]$, poiché $L^C \in \text{coDTIME}[f(n)]$, allora $L \in \text{coDTIME}[f(n)]$, ossia $\text{DTIME}[f(n)] \subseteq \text{coDTIME}[f(n)]$.
  Questi ultimi due punti dimostrano che $\text{DTIME}[f(n)] = \text{coDTIME}[f(n)]$. □

Teorema 6.12

Per ogni coppia di funzioni $f : \mathbb{N} \to \mathbb{N}$ e $g : \mathbb{N} \to \mathbb{N}$ tali che $\exists n_0 \in \mathbb{N} \ : \ \forall n \geq n_0$ allora

$f(n) \leq g(n)$ - ossia $f(n) \leq g(n)$ *definitivamente*

$$\begin{gathered} \mathrm{DTIME}[f(n)] \subseteq \mathrm{DTIME}[g(n)] \\ \mathrm{NTIME}[f(n)] \subseteq \mathrm{NTIME}[g(n)] \\ \mathrm{DSPACE}[f(n)] \subseteq \mathrm{DSPACE}[g(n)] \\ \mathrm{NSPACE}[f(n)] \subseteq \mathrm{NSPACE}[g(n)] \end{gathered}$$

Dim.
Sia $L \subseteq \{0,1\}^*$ tale che $L \in \mathrm{DTIME}[f(n)]$: allora esiste una macchina di Turing deterministica T che decide L e tale che $\mathrm{dtime}(T, x) \in O(f(|x|)) \subseteq O(g(|x|))$. Questo significa che $L \in \mathrm{DTIME}[g(n)]$. $\square$

Questo teorema ci dice che, se collochiamo un linguaggio L in una classe di complessità $\mathrm{DTIME}[f(n)]$, allora L appartiene anche a tutte le classi $\mathrm{DTIME}[g(n)]$ tali che $f(n) \leq g(n)$ definitivamente.

Teorema 6.13 (Gap Theorem)
Esiste una funzione totale e calcolabile $f : \mathbb{N} \to \mathbb{N}$ tale che

$$\mathrm{DTIME}[2^{f(n)}] \subseteq \mathrm{DTIME}[f(n)]$$

Dagli ultimi due teoremi visti, osserviamo che, se collochiamo un linguaggio L in $\mathrm{DTIME}[f(n)]$, allora L appartiene a **tutte** le classi $\mathrm{DTIME}[f(n)^k]$ per ogni $k \in \mathbb{N}$, per il teorema 6.12 (infatti $f(n) \leq f(n)^k$ definitivamente). Quindi abbiamo una gerarchia infinita di classi di complessità

$$\mathrm{DTIME}[f(n)] \subseteq \mathrm{DTIME}[f(n)^2] \subseteq \ldots \mathrm{DTIME}[f(n)^k]$$

e sempre per il teorema 6.12, data una funzione f totale e calcolabile ed una funzione g tale che $f(n) \leq g(n)$ definitivamente, allora

$$\mathrm{DTIME}[f(n)] \subseteq \mathrm{DTIME}[g(n)]$$

D'altra parte, nella definizione di una teoria della complessità in grado di classificare i linguaggi in classi di complessità crescente, sarebbe preferibile che $\mathrm{DTIME}[f(n)]$ **NON fosse contenuto in $\mathrm{DTIME}[g(n)]$ quando $f(n)$ è molto più grande di $g(n)$**, ad esempio quando $f(n) = 2^{g(n)}$, ma abbiamo visto che questo è contraddetto dal **Gap Theorem**.

Funzioni time/space-constructible

In questo paragrafo sono introdotte delle funzioni totali e calcolabili che, per essere calcolate, utilizzano quantità di risorse (tempo di calcolo o spazio di memoria) proporzionali al loro valore.

Def 6.1
Una funzione totale e calcolabile $f : \mathbb{N} \to \mathbb{N}$ è *time-constructible* se esiste una macchina di Turing T di tipo trasduttore che, preso in input un intero n espresso in unario, scrive sul nastro di output il valore $f(n)$ in unario e $\text{dtime}(T, n) \in O(f(n))$.

Def. 6.2
Una funzione totale e calcolabile $f : \mathbb{N} \to \mathbb{N}$ è *space-constructible* se esiste una macchina di Turing T di tipo trasduttore che, preso in input un intero n espresso in unario, scrive sul nastro di output il valore $f(n)$ in unario e $\text{dspace}(T, n) \in O(f(n))$.

Queste funzioni possono essere calcolate in tempo e spazio proporzionale al suo valore.
Sono time/space-constructible tutte le funzioni "regolari", come ad esempio le funzioni polinomiali e le funzioni esponenziali del tipo $2^{f(n)}$ dove $f(n)$ è una funzione time/space-constructible.

La funzione $f(n)$ che si trova nella dimostrazione del **gap theorem** è totale e calcolabile, ma non time-constructible.

È possibile mostrare che quando una funzione time-constructible f cresce molto più velocemente di una funzione g, la classe $\text{DTIME}[g(n)]$ è contenuta strettamente nella classe $\text{DTIME}[f(n)]$.
In effetti sussistono i seguenti teoremi di gerarchia:

Teorema 6.14 Teorema di gerarchia spaziale
Siano $f : \mathbb{N} \to \mathbb{N}$ e $g : \mathbb{N} \to \mathbb{N}$ due funzioni tali che f è space-constructible e

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Allora, $\text{DSPACE}[g(n)] \subset \text{DSPACE}[f(n)]$, ossia, esiste un linguaggio L tale che $L \in \text{DSPACE}[f(n)]$ e $L \notin \text{DSPACE}[g(n)]$.

Teorema 6.15 Teorema di gerarchia temporale
Siano $f : \mathbb{N} \to \mathbb{N}$ e $g : \mathbb{N} \to \mathbb{N}$ due funzioni tali che f è space-constructible e

$$\lim_{n\to\infty} \frac{g(n)\log(g(n))}{f(n)} = 0$$

Allora, $\text{DTIME}[g(n)] \subset \text{DTIME}[f(n)]$, ossia, esiste un linguaggio L tale che $L \in \text{DTIME}[f(n)]$ e $L \notin \text{DTIME}[g(n)]$.

Quindi, il teorema di gerarchia temporale ci dice che quando f è time-constructible $\mathrm{DTIME}[f(n)]$ **non è contenuto** in $\mathrm{DTIME}[g(n)]$ quando $f(n)$ è molto più grande di $g(n)$ - ad esempio, quando $f(n) = 2^{g(n)}$.

Lezione 13 - Funzioni time e space-constructible e specifiche classi di complessità

Nelle lezioni precedenti abbiamo lasciato delle questioni in sospeso, come la questione della definizione delle classi di complessità non deterministiche, dove viene richiesta l'accettabilità di un linguaggio, pur sapendo che, quando utilizziamo la quantità massima di risorse utilizzabili, un linguaggio accettabile è anche decidibile, non conosciamo la quantità di risorse che occorrono per rigettare le parole che non appartengono al linguaggio.
Inoltre sapiamo che, tutto ciò che può essere deciso da una macchina non deterministica può essere deciso anche da una macchina deterministica, ma un linguaggio $L \in \mathrm{NTIME}[f(n)]$ non sappiamo ancora in quale classe di complessità deterministica collocarlo e neanche sappiamo se l'appartenenza ad $\mathrm{NTIME}[f(n)]$ ci fornisca strumenti per affermare l'appartenenza a $\mathrm{DTIME}[\mathrm{altra\ funzione}]$.

Non è molto piacevole ammettere che se un linguaggio L è in $\mathrm{NTIME}[f(n)]$, e quindi sappiamo che esiste una macchina di Turing non deterministica NT che accetta le sue parole $x \in L$ eseguendo $O(f(|x|))$ istruzioni, ma non sappiamo quanto tempo ci occorre per capire che una parola non appartiene al linguaggio L, ovvero quando $x \notin L$ non sappiamo quante istruzioni sono eseguite da ciascuna computazione deterministica di NT che rigetta.

Teorema 6.16
Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione *time-constructible*. Se $L \in \mathrm{NTIME}[f(n)]$, allora L è decidibile in tempo non deterministico in $O(f(n))$.
Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione *space-constructible*. Se $L \in \mathrm{NTIME}[f(n)]$, allora L è decidibile in spazio non deterministico in $O(f(n))$.

Dim.
Sia $L \subseteq \Sigma^*$ tale che $L \in \mathrm{NTIME}[f(n)]$. Allora esiste una macchina di Turing non deterministica NT che accetta L e tale che, per ogni $x \in L$, $\mathrm{ntime}(NT, x) \leq c \cdot f(|x|)$ per qualche costante $c > 0$.
Poiché f è time-constructible, anche cf lo è, allora esiste una macchina di Turing di tipo trasduttore T_f tale che, per ogni $n \in \mathbb{N}$, $T_f(1^n)$ termina con il valore $cf(n)$ scritto sul nastro di output in unario dopo aver eseguito $O(cf(n))$ istruzioni.
Costruiamo ora, a partire da T_f e NT, la macchina NT' che decide L, quindi per ogni $x \in \Sigma^*$

- $NT^{'}(x)$ scrive $|x|$ in unario sul secondo nastro e invoca $T_f(|x|)$ e al termine della computazione troverà sul terzo nastro $cf(|x|)$ in unario.
- $NT^{'}(x)$ invoca $NT(x)$ e, per ogni quintupla eseguita non deterministicamente da $NT(x)$ nel seguente modo. Se il terzo nastro contiene un 1 allora $NT^{'}$ lo cancella e se $NT(x)$ accetta allora anche $NT^{'}(x)$ accetta e se $NT(x)$ rigetta allora anche $NT^{'}(x)$ rigetta. Se il terzo nastro di $NT^{'}$ è vuoto e $NT(x)$ non ha ancora terminato, allora $NT^{'}(x)$ rigetta.
  Osserviamo che le computazioni di $NT^{'}$ terminano sempre, infatti se la simulazione di una computazione di $NT(x)$ dura più di $cf(|x|)$ passi, la interrompiamo. Poi $NT^{'}$ decide L, infatti:
- se $x \in L$, allora $NT(x)$ accetta in al più $cf(|x|)$ passi e quindi $NT^{'}(x)$ accetta
- se $x \notin L$ allora o $NT(x)$ rigetta in al più $cf(|x|)$ passi e quindi $NT^{'}(x)$ rigetta, oppure $NT(x)$ non termina entro $cf(|x|)$ passi e quindi $NT^{'}(x)$ interrompe la computazione e rigetta.
  Quanto impegna però $NT^{'}$ a decidere se $x \in L$ oppure no?
- $O(cf(|x|))$ per calcolare $cf(x)$ - perché cf è time-constructible
- altri $cf(|x|)$ passi per simulare $cf(|x|)$ passi di $NT(x)$ ovvero $O(f(|x|))$ passi
  Quindi possiamo concludere che L è decidibile in tempo non deterministico $O(f(n))$. $\square$

Le uniche relazioni tra classi deterministiche e non deterministiche sono quelle basate sull'osservazione che una macchina deterministica è una particolare macchina non deterministica, ovvero

$$\text{DTIME}[f(n)] \subseteq \text{NTIME}[f(n)] \ \ e \ \ \text{DSPACE}[f(n)] \subseteq \text{NSPACE}[f(n)]$$

Inoltre sappiamo che tutto ciò che è deciso da una macchina non deterministica può essere deciso anche da una macchina deterministica, ma un linguaggio che appartiene ad $\text{NTIME}[f(n)]$ non sappiamo in quale classe di complessità temporale deterministica collocarlo, perché non sappiamo se esiste una funzione $g(n)$ che cresce molto più velocemente di $f(n)$ tale che possiamo affermare che "se $L \in \text{NTIME}[f(n)]$ allora $L \in \text{DTIME}[g(n)]$", a meno che la funzione limite f della classe non sia una funzione *time-constructible*.

Teorema 6.17
Per ogni funzione *time-constructible* $f : \mathbb{N} \rightarrow \mathbb{N}$

$$\text{NTIME}[f(n)] \subseteq \text{DTIME}[2^{O(f(n))}]$$

Dim.
Sia $L \subseteq \Sigma^*$ tale che $L \in \text{NTIME}[f(n)]$; allora esistono una macchina di Turing non deterministica *NT che accetta L* e una costante h tale che per ogni $x \in L$, $\text{ntime}(NT, x \leq hf(|x|))$.

Poiché f è time-constructible, esiste una macchina di Turing di tipo trasduttore T_f che, con input la rappresentazione in unario di un numero intero n, calcola la rappresentazione in unario di $f(n)$ in tempo $O(f(n))$.
Inidchiamo con k il grado di non determinismo di NT e utilizziamo di nuovo la tecnica della simulazione per definire una macchina di Turing deterministica Tche simuli il comportamento di NT: con input x, T simula in successione *tutte* le computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$.
La macchina T con input x opera come descritto di seguito:

1. Simula la computazione $T_f(|x|)$: per ogni carattere di x scrive sui N_2 un carattere 1 e, in seguito, calcola $f(|x|)$ scrivendolo su N_3. Infine, concatena h volte il contenuto del nastro N_3 ottenendo il valore $hf(|x|)$
2. Simula, una alla volta, tutte le computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$ utilizzando, per ciascuna computazione, la posizione della testina sul nastro N_3 come contatore, ovvero:
   - simula al più $hf(|x|)$ passi della computazione più a sinistra di tutte nell'albero $NT(x)$, se la computazione accetta entro $hf(x)$ passi allora T termina in q_A, altrimenti
   - simula al più $hf(|x|)$ passi della computazione immediatamente più a destra di quella appena simulata, se la computazione accetta entro $hf(x)$ passi allora T termina in q_A, altrimenti
   - $\vdots$
   - simula al più $hf(|x|)$ passi della computazione più a destra di tutte nell'albero $NT(x)$, se la computazione accetta entro $hf(x)$ passi allora T termina in q_A, altrimenti T termina in q_R

   Poiché, se $x \in L$, $\text{ntime}(NT, x) \leq hf(|x|)$, allora o in $hf(|x|)$ passi $NT(x)$ termina nello stato di accettazione oppure $x \notin L$. Quindi, se dopo aver simulato tutte le computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$, T non ha raggiunto lo stato di accettazione, allora può correttamente entrare nello stato di rigetto. Questo prova che T decide L.
   Ma quanto tempo impiega T a decidere L?
   Intanto la fase 1 della simulazione richiede $O(hf(|x|))$ passi, perché f è time-constructible, per la fase 2, detto k il grado di non determinismo di NT, il numero di computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$ è $k^{hf(|x|)}$ e ciascuna di esse viene simulata da T in $O(hf(|x|))$ passi. Allora

   $$\text{dtime}(T, x) \in O(hf(|x|)) + hf(|x|)k^{hf(|x|)} = O(hf(|x|)k^{hf(|x|)}) \subseteq O(2^O(f(|x|)))$$

   Infine, per il *teorema 6.3* (sulle dispense nel paragrafo 6.2), esiste una macchina T_1 ad un nastro tale che, per ogni input x, $o_{T_1}(x) = o_T(x)$ e

$$\text{dtime}(T_1, x) \leq \text{dtime}(T, x)^c \subseteq O(2^{O(f(|x|))})$$

Questo conclude la dimostrazione che $L \in \text{DTIME}[2^{O(f(|x|))}]$. $\square$

Specifiche classi di complessità

Siamo pronti ad introdurre le più rilevanti classi di complessità, definite sulla base di funzioni *time/space-constructible*.

- $\mathbf{P} = \bigcup_{k \in \mathbb{N}} \text{DTIME}[n^k]$: è la classe dei linguaggi decidibili in *tempo deterministico polinomiale*
- $\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}[n^k]$: è la classe dei linguaggi accettabili in *tempo non deterministico polinomiale*
- $\mathbf{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{DSPACE}[n^k]$: è la classe dei linguaggi decidibili in *spazio deterministico polinomiale*
- $\mathbf{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}[n^k]$: è la classe dei linguaggi accettabili in *spazio deterministico non polinomiale*
- $\mathbf{EXPTIME} = \bigcup_{k \in \mathbb{N}} \text{DTIME}[2^{p(n,k)}]$: è la classe dei linguaggi decidibili in *tempo deterministico esponenziale* ove l'esponente che descrive la funzione limite è un polinomio in n di grado k, indicato come $p(n, k)$
- $\mathbf{NEXPTIME} = \bigcup_{k \in \mathbb{N}} \text{NTIME}[2^{p(n,k)}]$: è la classe dei linguaggi accettabili in *tempo non deterministico esponenziale* ove l'esponente che descrive la funzione limite è un polinomio in n di grado k, indicato come $p(n, k)$
  Infine, una classse di complessità di funzioni: la classe delle *funzioni (totali) calcolabili in tempo deterministico polinomiale*

$$\mathbf{FP} = \bigcup_{k \in \mathbb{N}} \Big\{ f : \Sigma_1^* \to \Sigma_2^* : \text{esiste una macchina di Turing deterministica trasduttore } T \\ \text{che calcola } f \text{ e, per ogni } x \in \Sigma_1^*, \text{dtime}(T, x) \in O(|x|^k) \Big\}$$

Corollario 6.2
Valgono le seguenti relazioni di inclusione:

- $\mathbf{P} \subseteq \mathbf{NP}$, $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$, e $\mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME}$, conseguenza diretta del teorema 6.8: una macchina deterministica è una macchina non deterministica con grado di non determinismo 1
- $\mathbf{P} \subseteq \mathbf{PSPACE}$ e $\mathbf{NP} \subseteq \mathbf{NPSPACE}$, conseguenza diretta del teorema 6.9: per ogni funzione totale e calcolabile f

$$\text{DTIME}[f(n)] \subseteq \text{DSPACE}[f(n)] \; e \; \text{NTIME}[f(n)] \subseteq \text{NSPACE}[f(n)]$$

- $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$ e $\mathbf{NPSPACE} \subseteq \mathbf{NEXPTIME}$, sono conseguenza diretta del **teorema 6.10**: per ogni funzione totale e calcolabile f

$$\text{DSPACE}[f(n)] \subseteq \text{DTIME}[2^{O(1)f(n)}] \;\; e \;\; \text{NSPACE}[f(n)] \subseteq \text{NTIME}[2^{O(1)f(n)}]$$

- $\mathbf{NP} \subseteq \mathbf{EXPTIME}$ conseguenza diretta del **teorema 6.17**: per ogni funzione time-constructible f

$$\text{NTIME}[f(n)] \subseteq \text{DTIME}[2^{O(f(n))}]$$

  e i polinomi sono funzioni time-constructible.
  Tutte le relazioni fra classi di complessità dimostrate fino ad ora sono **inclusioni improprie**, ovvero per ciascuna di quelle relazioni non siamo in grado di dimostrare né l'inclusione propria né la coincidenza delle due classi che la costituiscono, ad esmepio sappiamo che tutti i linguaggi in $\mathbf{PSPACE}$ sono anche in $\mathbf{EXPTIME}$ e tutti i linguaggi che sono in $\mathbf{P}$ sono anche in $\mathbf{NP}$, ma non sappiamo rispondere alla domanda "tutti i linguaggi in $\mathbf{EXPTIME}$ sono anche in $\mathbf{PSPACE}$?" (e di conseguenza $\mathbf{PSPACE} = \mathbf{EXPTIME}$).
  Si tratta quindi di relazioni deboli, e sarebbe tremendo se si dimostrasse che tutte quelle inclusioni improprie fossero, invece, delle uguaglianze, perché non saremmo in grado di classificare i problemi "facili" e "difficili".

Uno strumento che dimostra l'inclusione stretta fra classi di complessità è il teorema di gerarchia temporale e come sua conseguenza vale il seguente teorema

Teorema 6.18

$$\mathbf{P} \subset \mathbf{EXPTIME}$$

Dim. sulle dispense per curiosità

Esiste anche un teorema che va nella direzione opposta, ovvero che dimostra l'uguaglianza di due classi, una deterministica e una non deterministica

Teorema 6.19

$$\mathbf{PSPACE} = \mathbf{NPSPACE}$$

Dim. sulle dispense per curiosità

Lezione 14 - Riduzioni polinomiali

Come già affermato nella lezione 13, le relazioni viste fra le classi di complessità sono *inclusioni improprie*, ad eccezione per $\mathbf{P} \subset \mathbf{EXPTIME}$ e $\mathbf{PSPACE} = \mathbf{NPSPACE}$. A parte queste due, per le altre relazioni non siamo in grado di dimostrare né l'inclusione propria, né la coincidenza.
Pur riuscendo a dimostrare che una certa classe di complessità $\mathcal{C}_1$ è contenuta propriamente in un'altra classe di complessità $\mathcal{C}_2$, ovvero $\mathcal{C}_1 \subset \mathcal{C}_2$, anche in questo caso, se dimostriamo che un linguaggio L appartiene a $\mathcal{C}_2$, come facciamo a sapere se quel linguaggio è anche in $\mathcal{C}_1$, oppure se, invece, è un linguaggio separatore fra $\mathcal{C}_1$ e $\mathcal{C}_2$, ossia $L \in \mathcal{C}_2 - \mathcal{C}_1$?
Sarebbe utile se avessimo, quindi, uno strumento che permettesse di individuare i **linguaggi separatori** fra due classi di complessità $\mathcal{C}_1$ e $\mathcal{C}_2$.

π-riduzioni

Utilizzando una nozione collegata al concetto di riducibilità funzionale, è possibile individuare i linguaggi *candidati* ad essere separatori fra due classi, e la nozione è quella di linguaggio **completo** per una data classe.

Alla definizione di riduzione, aggiungiamo una piccola richiesta.
Sia π un *predicato* definito sull'insieme delle funzioni totali e calcolabili, ovvero una proprietà che deve essere posseduta da una funzione, ad esempio:

- per ogni $x \in \Sigma^*,\ |f(x)| = |x|$
- per ogni $x \in \Sigma^*$, f è calcolabile in tempo polinomiale in $|x|$

Def. π-riduzione
Dati due linguaggi $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$, diciamo che **L_1 è π-riducibile ad L_2** e scriviamo $L_1 \preceq_\pi L_2$ se esiste una funzione $f : \Sigma_1^* \to \Sigma_2^*$ tale che:

- f è totale e calcolabile, ossia
  - è definita per ogni $x \in \Sigma_1^*$
  - esiste una macchina di Turing T_f di tipo trasduttore 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
- per ogni $x \in \Sigma_1^*$ vale che $x \in L_1 \iff f(x) \in L_2$
- f soddisfa π

Lo strumento che potrebbe permettere di individuare i linguaggi separatori fra due classi di complessità è basato sui seguenti due concetti che si riferiscono alle π-riduzioni:

- **chiusura** di una classe rispetto a una π-riduzione
- **completezza** di un linguaggio per una classe rispetto a una π-riduzione

Def. chiusura
Una classe di complessità $\mathcal{C}$ è *chiusa* rispetto ad una generica π-riduzione se, per ogni coppia di linguaggi L_1 e L_2 tali che $L_1 \preceq_\pi L_2$ e $L_2 \in \mathcal{C}$, si ha che $L_1 \in \mathcal{C}$.

La chiusura di una classe $\mathcal{C}$ rispetto ad una π-riduzione può essere utilizzata per dimostrare l'appartenenza di un linguaggio L a $\mathcal{C}$:

- segue direttamente dalla definizione, che, se sappiamo che una classe di complessità $\mathcal{C}$ è chiusa rispetto ad una π-riduzione e che un certo linguaggio L_0 appartiene a $\mathcal{C}$, allora, se dimostriamo che $L \preceq_\pi L_0$, possiamo dedurre che anche L appartiene a $\mathcal{C}$.

Def. completezza
Sia $\mathcal{C}$ una classe di complessità di linguaggi e sia $\preceq_\pi$ una generica π-riduzione. Un linguaggio $L \subseteq \Sigma^*$ è $\mathcal{C}$-completo rispetto alla π-riducibilità se:

- $L \in \mathcal{C}$
- per ogni altro $L^{'} \in \mathcal{C}$, vale che $L^{'} \preceq_\pi L$

Le ultime due definizioni sono gli strumenti che ci permettono di arrivare al concetto di linguaggio "più difficile" in una classe.

> **Esempio**
> Abbiamo due classi di complessità $\mathcal{C}_1$ e $\mathcal{C}_2$ tali che $\mathcal{C}_1 \subseteq \mathcal{C}_2$ e sappiamo che $\mathcal{C}_1$ è chiusa rispetto ad una qualche π-riduzione, allora, per ogni coppia di linguaggi L_1 e L_2, tali che $L_1 \preceq_\pi L_2$ e $L_2 \in \mathcal{C}_1$, si ha che $L_1 \in \mathcal{C}_1$.
> Se per caso trovassimo un linguaggio L che è $\mathcal{C}_2$-completo rispetto a $\preceq_\pi$, ossia $L \in \mathcal{C}_2$ e per ogni altro $L_0 \in \mathcal{C}_2$, vale che $L_0 \preceq_\pi L$ e se dimostriamo che $L \in \mathcal{C}_1$, abbiamo che per ogni altro $L_0 \in \mathcal{C}_2$ vale che $L_0 \preceq_\pi L$ e inoltre $L \in \mathcal{C}_1$.
> Allora in virtù della chiusura di $\mathcal{C}_1$ rispetto alla π-riduzione, per ogni altro $L_0 \in \mathcal{C}_2$ vale che $L_0 \in \mathcal{C}_1$ e dunque, $\mathcal{C}_1 = \mathcal{C}_2$.

Riassumendo, abbiamo due classi di complessità $\mathcal{C}_1$ e $\mathcal{C}_2$, tali che $\mathcal{C}_1 \subseteq \mathcal{C}_2$ e sappiamo che $\mathcal{C}_1$ è chiusa rispetto ad una qualche π-riduzione. Se per caso trovassimo un linguaggio L che è $\mathcal{C}_2$-completo rispetto a $\preceq_\pi$ allora:

- da un ipotetico algoritmo che decide L utilizzando una quantità di risorse pari a quella che definisce la classe $\mathcal{C}_1$ - cioé se dimostrassimo che $L \in \mathcal{C}_1$ - potremmo dedurre un algoritmo

che decide qualunque problema in $\mathcal{C}_2$ utilizzando una quantità di risorse pari a quella che definisce la classe $\mathcal{C}_1$
Allora se riuscissimo a dimostrare che $L \in \mathcal{C}_1$ sapremmo automaticamente che tutti i linguaggi in $\mathcal{C}_2$ sono anche in $\mathcal{C}_1$ - ossia sapremmo che $\mathcal{C}_1 = \mathcal{C}_2$.
Ma possiamo vederla anche in un altro modo: **se $\mathcal{C}_1 \subseteq \mathcal{C}_2$ e L è $\mathcal{C}_2$-completo** e se qualcuno riuscisse a dimostrare che $\mathcal{C}_1 \neq \mathcal{C}_2$ allora sapremmo automaticamente che $L \notin \mathcal{C}_1$.
L sarebbe un linguaggio "più difficile" fra tutti i linguaggi in $\mathcal{C}_2$

Teorema 6.20
Siano $\mathcal{C}$ e $\mathcal{C}_0$ due classi di complessità tali che $\mathcal{C}_0 \subseteq \mathcal{C}$.
Se $\mathcal{C}_0$ è chiusa rispetto una π-riduzione allora, per ogni linguaggio L che sia $\mathcal{C}$-completo rispetto a $\preceq_\pi$, $L \in \mathcal{C}_0$ se e solo se $\mathcal{C} = \mathcal{C}_0$.

Dim.

- Banalmente, se $\mathcal{C} = \mathcal{C}_0$, poiché L è $\mathcal{C}$-completo e, dunque $L \in \mathcal{C}$, allora $L \in \mathcal{C}_0$
- viceversa, supponiamo che $L \in \mathcal{C}_0$. Poiché L è $\mathcal{C}$-completo rispetto a $\preceq_\pi$, allora, per ogni $L^{'} \in \mathcal{C}$, $L^{'} \preceq_\pi L$. Poiché $\mathcal{C}_0$ è chiusa rispetto a $\preceq_\pi$, allora, per ogni $L^{'} \in \mathcal{C}$, $L^{'} \in \mathcal{C}_0$ e quindi $\mathcal{C} = \mathcal{C}_0$ $\square$

Introduciamo una particolare π-riduzione il cui predicato π specifica il costo computazionale del calcolo della funzione

Def. riducibilità polinomiale
Siano $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$ due linguaggi; diciamo che L_1 è **polinomialmente riducibile** a L_2 e scriviamo $L_1 \preceq_p L_2$, se esiste una funzione totale e calcolabile $f : \Sigma_1^* \to \Sigma_2^*$ tale che

- $f \in \mathbf{FP}$ (f è totale e calcolabile in tempo polinomiale), ossia
  - è definita per ogni parola $x \in \Sigma_1^*$
  - 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
  - esiste una costante c tale che per ogni $x \in \Sigma_1^*$, $\text{dtime}(T_f, x) \in O(|x|^c)$
- per ogni $x \in \Sigma_1^*$ vale che $\left[x \in L_1 \iff f(x) \in L_2\right]$

  Oss.
  Siccome d'ora in poi faremo sempre riferimento alla riducibilità polinomiale, per semplificare le nozioni, scriveremo solo $L_1 \preceq L_2$ per indicare che L_1 è riducibile polinomialmente a L_2.

> **Esempio**
> Abbiamo due linguaggi $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$ e riusciamo a dimostrare che $L_1 \preceq L_2$, cioé dimostriamo che esistono un traduttore T_r e una costante c tali che, per ogni $x \in \Sigma_1^*$ e inoltre, per ogni $x \in \Sigma_1^*$, $\text{dtime}(T_r, x) \in O(|x|^c)$.
> Supponiamo di sapere che $L_2 \in \text{DTIME}[f(n)]$, cioé esiste un riconoscitore T_2 tale che, per

> ogni $y \in \Sigma_2^*$, T_2 accetta se e soltanto se $y \in L_2$ e, inoltre, per ogni $y \in \Sigma_2^*$, $\mathrm{dtime}(T_2, y) \in O(f|y|)$.
> Allora possiamo costruire la seguente macchina T_1: con input $x \in \Sigma_1^*$, T_1 opera in due fasi utilizzando due nastri
>
> 1. T_1 simula $T_r(x)$ scrivendo l'output y sul secondo nastro
> 2. T_1 simula $T_2(y)$ sul secondo nastro; se $T_2(y)$ accetta allora anche T_1 accetta, se $T_2(y)$ rigetta, allora anche T_1 rigetta
> T_1 quindi decide L_1, perché $T_2(y)$ accetta se e solo se $y \in L_2$ e $y \in L_2$ se e solo se $x \in L_1$
>   Ma quanto impiega T_1 a decidere L_1?
>   Con input x la prima fase termina in $O(|x|^c)$ passi e la seconda fase termina in $O(f(|y|))$ passi. Ma quanto è grande $|y|$ in funzione di $|x|$? Poiché $T_r(x)$ impiega $O(|x|^c)$ passi per calcolare y, e in questo numero di passi sono conteggiati anche i passi che occorono a scrivere y sul nastro di output, allora $|y| \in O(|x|^c)$ e quindi per ogni $x \in \Sigma_1^*$, $T_1(x)$ termina in $O(|x|^c + f(|x|^c))$ passi, ossia $L_1 \in \mathrm{DTIME}[n^c + f(n^c)]$.

In particolare: abbiamo due linguaggi $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$ e sappiamo che $L_1 \preceq L_2$. Abbiamo appena dimostrato che se $L_2 \in \mathbf{P}$ allora $L_1 \in \mathbf{P}$, infatti, in questo caso, esiste una costante k tale che $L_2 \in \mathrm{DTIME}[n^k]$, allora da quanto visto nell'esempio $L_1 \in \mathrm{DTIME}[n^c + (n^c)^k] \subseteq \mathbf{P}$.

Tutte le classi di complessità introdotte nella lezione 13 sono chiuse rispetto alla riducibilità polinomiale.

Teorema 6.21
La classe $\mathbf{P}$ è chiusa rispetto alla riducibilità polinomiale

Questo teorema dimostra solo il caso visto nell'esempio precedente, quindi "se $L_1 \preceq L_2$ e $L_2 \in \mathbf{P}$ allora $L_1 \in \mathbf{P}$"

Allo stesso modo si dimostra che quando $L_1 \preceq L_2$ se $L_2 \in \mathbf{EXPTIME}$ allora $L_1 \in \mathbf{EXPTIME}$ e così per le altre classi di complessità.

Teorema 6.22
Le classi $\mathbf{NP}$, $\mathbf{PSPACE}$, $\mathbf{EXPTIME}$, $\mathbf{NEXPTIME}$ sono chiuse rispetto alla riducibilità polinomiale.

Linguaggi NP-completi

Def.
Un linguaggio $L \subseteq \Sigma^*$ è $\mathbf{NP}$-completo rispetto alla riducibilità polinomiale se:

- $L \in \mathbf{NP}$
- per ogni altro $L_0 \in \mathbf{NP}$ vale che $L_0 \preceq L$

I linguaggi $\mathbf{NP}$-completi sono particolarmente importanti per il loro rualo di possibili *linguaggi separatori* fra le classi $\mathbf{P}$ e $\mathbf{NP}$.

Corollario
Se $\mathbf{P} \neq \mathbf{NP}$ allora, per ogni linguaggio $\mathbf{NP}$-completo L, $L \notin \mathbf{P}$

Dim.
Supponiamo che L sia un linguaggio $\mathbf{NP}$-completo e che $L \in \mathbf{P}$.
Poiché L è $\mathbf{NP}$-completo allora, per ogni linguaggio $L_0 \in \mathbf{NP}$, $L_0 \preceq L$, ma se $L \in \mathbf{P}$, poiché $\mathbf{P}$ è chiusa rispetto a $\preceq$, questo implica che, per ogni $L_0 \in \mathbf{NP}$, $L_0 \in \mathbf{P}$, ossia $\mathbf{P} = \mathbf{NP}$, contraddicendo l'ipotesi. $\square$

Qual è il senso di questo corollario? Intanto diciamo che è molto improbabile che un linguaggio $\mathbf{NP}$-completo appartenga a $\mathbf{P}$, perché si sospetta che $\mathbf{P} \neq \mathbf{NP}$ secondo la **congettura fondamentale della complessità computazionale**.
Quindi se vogliamo dimostrare che non esiste un algoritmo deterministico che decide in tempo polinomiale un linguaggio che è in $\mathbf{NP}$, allora dobbiamo dimostrare che quel linguaggio è $\mathbf{NP}$-completo. Se, invece, abbiamo un linguaggio $\mathbf{NP}$-completo e progettiamo un algortimo *deterministico* che decide quel linguaggio in tempo polinomiale le opzioni sono due:

- abbiamo risolto la congettura
- abbiamo sbagliato qualcosa

Nel campo ddella calcolabilità, le riduzioni si rivelano utili tanto per dimostrare che un linguaggio è accettabile/decidibile, quanto per dimostrare che un linguaggio non è accettabile/decidibile. Dato un linguaggio L_1:

- se dimostro che $L_1 \preceq L_2$, per un qualche altro linguaggio L_2 decidibile, allora posso concludere che anche L_1 è decidibile
- se dimostro che $L_0 \preceq L_1$, per un qualche linguaggio L_0 non decidibile, allora posso concludere che anche L_1 è non decidibile

Allo stesso modo, le riduzioni polinomiali sono uno strumento utile tanto per dimostrare che un linguaggio è in $\mathbf{P}$, quanto per dimostrare che un linguaggio *probabilmente* non è in $\mathbf{P}$. Dato un linguaggio L_1:

- se dimostro che $L_1 \preceq L_2$ per un qualche altro linguaggio $L_2 \in \mathbf{P}$, allora posso concludere che anche $L_1 \in \mathbf{P}$
- se dimostro che $L_0 \preceq L_1$, per un qualche altro linguaggio L_0, allora posso concludere che ** L_1 non può essere più facile di L_0,

- ovvero se L_0 probabilmente non appartiene a $\mathbf{P}$ allora anche L_1 probabilmente non appartiene a $\mathbf{P}$.

Lezione 15 - Classi Complemento

Nella lezione 13 abbiamo introdotto le specifiche classi di complessità. Consideriamo i suoi complementi

$$co\mathbf{P} = \{L \subset \{0,1\}^* : L^c \in \mathbf{P}\}$$
$$co\mathbf{NP} = \{L \subset \{0,1\}^* : L^c \in \mathbf{NP}\}$$

e allo stesso modo, anche le classi
$co\mathbf{EXPTIME},\ co\mathbf{NEXPTIME},\ co\mathbf{PSPACE}$

In generale definiamo anche

$$\begin{aligned}
\text{coDTIME}[f(n)] &= \left\{L \subset \{0,1\}^* : L^c \in \text{DTIME}[f(n)]\right\} \\
\text{coSPACE}[f(n)] &= \left\{L \subset \{0,1\}^* : L^c \in \text{DSPACE}[f(n)]\right\} \\
\text{coNTIME}[f(n)] &= \left\{L \subset \{0,1\}^* : L^c \in \text{NTIME}[f(n)]\right\} \\
\text{coNSPACE}[f(n)] &= \left\{L \subset \{0,1\}^* : L^c \in \text{NSPACE}[f(n)]\right\}
\end{aligned}$$

Oss.
Nella definizione delle **classi di complessità complemento** non viene specificato come vengono decisi o accettati i linguaggi che vi appartengono, ma viene specificato come vengono decisi o accettati i complementi dei linguaggi che appartengono.

Quando si parla di classi deterministiche in realtà fare questa differenziazione è irrilevante

Teorema 6.11
Per ogni funzione totale e calcolabile $f : \mathbb{N} \to \mathbb{N}$,

$$\text{DTIME}[f(n)] = \text{coDTIME}[f(n)] \ e \ \text{DSPACE}[f(n)] = \text{coDSPACE}[f(n)]$$

Dim.
Prendiamo una macchina T che decide L tale che, per ogni x, $\text{dtime}(T, x) \in O(f(|x|))$.
Si costruisce una nuova macchina T' complementando gli stati di accettazioni e di rigetto di T,

quindi aggiungendo le quintuple $\langle q_A, s, s, q'_R, f\rangle$ e $\langle q_R, s, s, q'_A, f\rangle$ per ogni $s \in \{0, 1, \square\}$, dove q'_A e q'_R sono gli stati di accettazione e di rigetto di $T^{'}$.
$T^{'}$ dedice L^c e $\text{dtime}(T', x) \in O(f(|x|))$. $\square$

Analogo per $\text{DSPACE}[f(n)] = \text{coDSPACE}[f(n)]$.

Dal **teorema 6.11** si deriva il seguente corollario

Corollario 6.3
$\mathbf{P} = co\mathbf{P}$, ma anche $co\mathbf{PSPACE} = \mathbf{PSPACE}$

La classe $co\mathbf{NP}$

Ricordiamo che $\mathbf{NP}$ è la classe dei linguaggi *accettati* in tempo polinomiale da una macchina non deterministica NT, quindi $co\mathbf{NP}$ è la classe dei linguaggi il cui complemento è accettato in tempo polinomiale da una macchina non deterministica, cioé

$$co\mathbf{NP} = \{L : L^c \in \mathbf{NP}\}$$

A questo punto potremmo pensare che, come $\mathbf{P}$ coincida col suo complemento $co\mathbf{P}$, allora anche $\mathbf{NP}$ e $co\mathbf{NP}$ coincidono.
Iniziamo ricordando l'asimmetria delle definizioni di accettazone e rigetto di una macchina non deterministica, infatti NT:

- **accetta** un input x se esiste una computazione deterministica in $NT(x)$ che termina in q_A
- **rigetta** un input x se ogni computazione deterministica in $NT(x)$ termina in q_R

Il **problema** quindi sta propio nelle asimmetrie delle definizioni di accettazione e rigetto.
Infatti, proviamo ad applicare la stessa tecnica utilizzata nel teorema 6.11 ad una macchina non deterministica NT

- Costruiamo una macchina NT' invertendo gli stati di accettazione e di rigetto di NT e vediamo se accetta il complemento del linguaggio accettato da NT
  Scegliamo un linguaggio $L \subseteq \{0, 1\}^*$ accettato da una macchina di Turing non deterministica NT.
  Ricordiamo che il linguaggio complemento di L è $L^c = \{0, 1\}^* - L$, ovvero, per ogni $x \in \{0, 1\}^*$
- Se $x \in L \implies x \notin L^c$
- Se $x \notin L \implies x \in L^c$
  Allora, una macchina non deterministica NT^c accetta L^c se, per ogni $x \in \{0, 1\}^*$,
- Se $x \in L \implies NT^c(x)$ non accetta

- Se $x \notin L \implies NT^c(x)$ accetta
  e quindi
- Se $x \in L \implies$ ogni computazione deterministica in $NT^c(x)$ **non** termina in q_A
- Se $x \notin L \implies$ esiste una computazione deterministica in $NT^c(x)$ che termina in q_A
  Prima di invertire gli stati di accettazione e di rigetto di NT, costruiamo una nuova macchina NT_1 che accetta L.
  Prendiamo NT ed aggiungiamo all'insieme delle sue quintuple, le quintuple $\langle q_0, s, s, q_R, f\rangle$ per ogni $s \in \{0, 1, \square\}$

> **Warning**
>
> Per ogni $x \in \{0,1\}^*$ **esiste sempre** una computazione deterministica di $NT_1(x)$ che termina in q_R.

“FI/MOD II/img/img10.png” non trovato.

NT_1 **accetta** L, infatti, per ogni $x \in L$: poiché NT accetta L, allora $NT(x)$ accetta e allora esiste una computazione deterministica di $NT(x)$ che termina in q_A, ma quella stessa computazione deterministica compare anche in $NT_1(x)$ e, quindi, $NT_1(x)$ accetta

“FI/MOD II/img/img12.png” non trovato.

D'altra parte, per ogni $x \notin L$: poiché NT accetta L, allora $NT(x)$ non accetta (rigetta o non termina) e allora non esiste alcuna computazione deterministica di $NT(x)$ che termina in q_A, e allora stesso modo non esiste in $NT_1(x)$ una computazione deterministica che accetta e, quindi, $NT_1(x)$ non accetta

“FI/MOD II/img/img13.png” non trovato.

Quindi abbiamo un linguaggio $L \subseteq \{0,1\}^*$ accettato dalla macchina non deterministica NT_1 e ora applichiamo la stessa tecnica utilizzata nella dimostrazione del teorema 6.11: costruiamo una nuova macchina NT_1^C invertendo gli stati di accettazione e rigetto di NT_1. Quello che ci aspettiamo è che NT_1^C accetti L^C, vediamo.
Scegliamo $x \in \{0,1\}^*$ e poniamo $x = x_1x_2\dots x_n$. Se $x \in L^C$:

- in $NT_1(x)$ esiste la computazione deterministica $\langle q_0, x_1, x_1, q_R, f\rangle$ che termina in q_R
- la stessa computazione deterministca compare anche in $NT_1^C(x)$ che invece termina in q_A.
  Quindi $NT_1^C(x)$ accetta.
  Se $x \notin L^C$:
- se fosse vero che NT_1^C decide L^C, allora $NT_1(x)^C$ dovrebbe non accettare.
- In $NT_1(x)$ esiste la computazione deterministica $\langle q_0, x_1, x_1, q_R, f\rangle$ che termina in q_R

- la stessa computazione deterministca compare anche in $NT_1^C(x)$ che invece termina in q_A. Quindi $NT_1^C(x)$ accetta, ma ricordiamoci che $x \notin L^C$ e quindi $NT_1(x)$ non dovrebbe accettre!
Quindi NT_1^C accetta qualunque sia x e allora NT_1^C non accetta L^C!

Quindi l'asimmetria delle definizioni di accettazione e rigetto nelle macchine non deterministiche non permette di derivare una macchina che decide L^C invertendo gli stati di accettazione e rigetto di una macchina non deterministica che decide L.

Quindi non possiamo affermare che $co\mathbf{NP} = \mathbf{NP}$. Ma tutto questo ragionamento ci permette di affermare che $co\mathbf{NP} \neq \mathbf{NP}$? No, perché la dimostrazione dell'uguaglianza potrebbe seguire una strada diversa da quella dell'inversione degli stati finali di una macchina non deterministica.

Abbiamo detto nelle lezioni passate che le relazioni fra le classi di complessità sono inclusioni deboli, nelle quali non si riesce a dimostrare né che le due classi sono diverse, né che sono uguali. Il caso più famoso riguarda le classi $\mathbf{P}$ e $\mathbf{NP}$. Sappiamo che $\mathbf{P} \subseteq \mathbf{NP}$ e quindi ogni problema in $\mathbf{P}$ è contenuto anche in $\mathbf{NP}$. Ma non sappiamo se $\mathbf{P} = \mathbf{NP}$, ossia se ogni problema in $\mathbf{NP}$ è contenuto in $\mathbf{P}$ e non sappiamo neanche se $\mathbf{P} \neq \mathbf{NP}$, ossia se esiste un problema in $\mathbf{NP}$ che non è contenuto in $\mathbf{P}$.

La **congettura fondamentale della teoria della complessità computazionale** ipotizza che $\mathbf{P} \neq \mathbf{NP}$ e ora abbiamo scoperto una nuova congettura, ovvero la **seconda congettura della teoria della complessità computazionale**, che ipotizza che $co\mathbf{NP} \neq \mathbf{NP}$.

Le due congetture non sono del tutto indipendenti, come descritto nel prossimo teorema.

Teorema 6.23
Se $\mathbf{P} = \mathbf{NP}$ allora $\mathbf{NP} = co\mathbf{NP}$

Dim.
Per il corollario 6.3 $\mathbf{P} = co\mathbf{P}$ e per l'ipotesi $\mathbf{P} = \mathbf{NP}$ e quindi $co\mathbf{P} = co\mathbf{NP}$ e allora $\mathbf{NP} = \mathbf{P} = co\mathbf{P} = co\mathbf{NP}$. $\square$

Il teorema afferma che se è falsa la **congettura fondamentale della teoria della complessità computazionale** allora è falsa anche la **seconda congettura della teoria della complessità computazionale**

Questo teorema può essere anche letto "se $\mathbf{NP} \neq co\mathbf{NP}$ allora $\mathbf{P} \neq \mathbf{NP}$", ovvero se è vera la **seconda congettura della teoria della complessità computazionale** allora è vera anche la **congettura fondamentale della teoria della complessità computazionale**.

Teorema 6.24
La classe $co\mathbf{NP}$ è chiusa rispetto alla riducibilità polinomiale.

Dim. analoga a quella del teorema 6.21.

Come per tutte le classi di complessità, anche per la classe $co\mathbf{NP}$ possiamo definire i linguaggi completi rispetto alla riducibilità polinomiale.

Def.
Un linguaggio è $co\mathbf{NP}$-completo se:

- $L \in co\mathbf{NP}$
- per ogni linguaggio $L^{'} \in co\mathbf{NP}$ si ha che $L^{'} \preceq L$

Come visto nella scorsa lezione, i linguaggi $\mathbf{NP}$-completi sono i possibili *linguaggi separatori* fra $\mathbf{P}$ e $\mathbf{NP}$, ossia, nell'ipotesi $\mathbf{P} \neq \mathbf{NP}$, **un linguaggio $\mathbf{NP}$-completo non può essere contenuto in $\mathbf{P}$**.

Ci proponiamo di fare la stessa cosa con la classe $co\mathbf{NP}$, quindi vogliamo dimostrare che i linguaggi $co\mathbf{NP}$-completi sono i candidati ad essere i linguaggi separatori fra $\mathbf{NP}$ e $co\mathbf{NP}$, ossia, nell'ipotesi che $\mathbf{NP} \neq co\mathbf{NP}$, un linguaggio $co\mathbf{NP}$-completo **non può essere contenuto** in $\mathbf{NP}$.

I prossimi due teoremi mirano proprio a questo obiettivo.

Teorema 6.25
Un linguaggio L è $\mathbf{NP}$-completo se e solo se il suo complemento L^c è $co\mathbf{NP}$-completo.

Dim.
$\Longrightarrow$ Sia L un linguaggio $\mathbf{NP}$-completo e mostriamo che L^C è un linguaggio $co\mathbf{NP}$-completo.

- $L \in \mathbf{NP}$ e quindi $L^C \in co\mathbf{NP}$
  Dobbiamo mostrare che per ogni $L_1 \in co\mathbf{NP}$ vale che $L_1 \preceq L^C$.
- Sia allora L_1 un qualsiasi linguaggio in $co\mathbf{NP}$, allora $L_1^C \in \mathbf{NP}$. Poiché L è completo per la classe $\mathbf{NP}$ allora, per ogni $L_0 \in \mathbf{NP}$, $L_0 \preceq L$, allora, in particolare, poiché $L_1^C \in \mathbf{NP}$, vale che $L_1^C \preceq L$. Questo significa che esiste una funzione $f_1 : \{0,1\}^* \rightarrow \{0,1\}^*$ tale che $f_1 \in \mathbf{FP}$ e, per ogni $x \in \{0,1\}^*$, $x \in L_1^C$ se e solo se $f_1(x) \in L$. Ma questo equivale a dire che, per ogni $x \in \{0,1\}^*$, $x \notin L_1^C$ se e solo se $f_1(x) \notin L$, ossia **per ogni $x \in \{0,1\}^*$, $x \in L_1$ se e soltanto se $f_1(x) \in L^C$** e quindi $L_1 \preceq L^C$.
  Poiché L_1 è un qualsiasi linguaggio in $co\mathbf{NP}$, questo dimostra che L^C è completo per $co\mathbf{NP}$.

$\Longleftarrow$ Sia L^C un linguaggio $co\mathbf{NP}$-completo e mostriamo che L è $\mathbf{NP}$-completo.

- $L^C \in co\mathbf{NP}$ e quindi $L \in \mathbf{NP}$
  Dobbiamo mostrare che, per ogni $L_1 \in \mathbf{NP}$, vale che $L_1 \preceq L$.
- Sia L_1 un qualsiasi linguaggio in $\mathbf{NP}$, allora $L_1^C \in co\mathbf{NP}$, poiché L^C è completo per $co\mathbf{NP}$, allora per ogni $L_0 \in co\mathbf{NP}$, $L_0 \preceq L^C$, in particolare, poiché $L_1^C \in co\mathbf{NP}$, vale che

$L_1^C \preceq L^C$. Questo significa che esiste una funzione $f_1 : \{0,1\}^* \to \{0,1\}^*$ tale che $f_1 \in \mathbf{FP}$ e, per ogni $x \in \{0,1\}^*$, $x \in L_1^C$ se e solo se $f_1(x) \in L^C$. Ma questo equivale a dire che, per ogni $x \in \{0,1\}^*$, $x \notin L_1^C$ se e solo se $f_1(x) \notin L^C$, ossia **per ogni $x \in \{0,1\}^*$, $x \in L_1$ se e soltanto se $f_1(x) \in L$**.
Poiché L_1 è un qualsiasi linguaggio in $\mathbf{NP}$, questo dimostra che L è completo per $\mathbf{NP}$. $\square$

Teorema 6.26
Se esiste un liguaggio L $\mathbf{NP}$-completo tale che $L \in co\mathbf{NP}$, allora $\mathbf{NP} = co\mathbf{NP}$.

Dim.
Dimostriamo il teorema mostrando prima che

1. $co\mathbf{NP} \subseteq \mathbf{NP}$
2. $\mathbf{NP} \subseteq co\mathbf{NP}$
   Sia L un qualunque linguaggio $\mathbf{NP}$-completo tale che $L \in co\mathbf{NP}$
3. Poiché $L \in co\mathbf{NP}$, allora $L^C \in \mathbf{NP}$. Poiché L è $\mathbf{NP}$-completo, per il teorema 6.25 L^C è $co\mathbf{NP}$-completo, quindi per ogni $L' \in co\mathbf{NP}$ si ha che $L' \preceq L^C$. Ma $\mathbf{NP}$ è chiusa rispetto alla riducibilità polinomiale e allora per ogni linguaggio $L' \in co\mathbf{NP}$ si ha che $L' \in \mathbf{NP}$.
   Questo dimostra che $co\mathbf{NP} \subseteq \mathbf{NP}$
4. Poiché L è $\mathbf{NP}$-completo allora, per ogni $L'' \in \mathbf{NP}$ si ha che $L'' \preceq L$, ma $L \in co\mathbf{NP}$ e inoltre $co\mathbf{NP}$ è chiusa rispetto alla riducibilità polinomiale, allora per ogni $L'' \in \mathbf{NP}$ si ha che $L'' \in co\mathbf{NP}$ e questo dimostra che $\mathbf{NP} \subseteq co\mathbf{NP}$.
   $\square$

Lezione 16 - Problemi e codifiche

Le teorie della complessità e della calcolabilità sono fondate sul concetto di **appartenza di una parola ad un insieme di parole**, un concetto:

- semplice
- elegante
- **formale**
- **rigoroso**

Ma nella vita reale ti capita mai di domandarti "ma questa parola apparterrà a questo insieme?", ossia capita di trover trovare soluzioni ad istanze di problemi. Allora queste teorie sarebbe bello trasferirle nel mondo dei problemi.
Ma "**trovare la soluzione ad una istanza di un problema**" è più arbitrario rispetto al concetto di appartenenza di una parola ad un insieme di parole. Cerchiamo di rendere il concetto di trovare la soluzione di un problema più formale.

Tipi di problemi

Come possiamo schematizzare un problema? Allora la struttura di un problema, qualunque esso sia è la seguente:
abbiamo un insieme di oggetti conosciuti - **i dati** - all'interno di un secondo insieme di oggetti - **le soluzioni possibili** - e dobbiamo cercare gli oggetti che soddifano certi vincoli, ovvero **le soluzioni** e sulla base di questa ricerca dobbiamo fornire una risposta.

> **Esempio problema generico**
> Dato $n \in \mathbb{N} \dots$ [domanda sui divisiori di n].
> Dobbiamo descrivere le istanze del problema e facciamolo definendo l'insieme delle istanze $\mathfrak{I}$.
> Un elemento di $\mathfrak{I}$ corrisponde ad un istanza di un problema. Nell'esempio $\mathfrak{I} = \mathbb{N}$.
> L'insieme delle soluzioni possibili descrive le soluzioni possibili per un istanza x del problema, definendo $S(x)$, ovvero tutti gli oggetti che dobbiamo testare per verificare se soddisfano i vincoli del problema. In questo caso $S(x) = \{y \in \mathbb{N} : y \leq x\}$.
> Dobbiamo ora cercare gli oggetti che soddisfano i vincoli del problema, quindi tutti gli oggetti, all'interno delle soluzioni possibili $S(x)$, che soddisfano le richieste del problema. Descriviamo tutte le soluzioni possibili associate ad un'istanza x che soddisfano i vincoli del problema definiendo l'insieme $\eta(S(x))$ di **soluzioni effettive** per l'istanza x. L'insieme $\eta(S(x))$ contiene

> gli oggetti che sono soluzioni possibili per x e che soddisfano i vincoli del problema, quindi $\eta(S(x)) = \{y \in S(x) : y \text{ è un divisore di } x\}$.
> Sulla base degli oggetti trovati forniamo una risposta, che definiamo con una funzione ρ che associa all'insieme delle soluzioni effettive per l'istanza x una risposta scelta nell'insieme R delle risposte.

> **Esempio 1**
> Dato $n \in \mathbb{N}$, **elencare tutti i divisori di n**.
> In questo caso $R = 2^{\mathbb{N}}$, ossia la risposta ad un'istanza del problema è un sottoinsieme di $\mathbb{N}$ e per ogni istanza n del problema $\rho(\eta(S(n))) = \eta(S(n))$.

> **Esempio 2**
> Dato $n \in \mathbb{N}$, verificare se n è primo.
> In questo caso $R = \{\text{vero}, \text{falso}\}$ e per ogni istanza n del problema
> $\rho(\eta(S(n))) = \Big[\eta(S(n)) = \{1, n\}\Big]$

> **Esempio 3**
> Dato $n \in \mathbb{N}$, **calcolare un divisore d non banale di n** (ossia $d > 1 \land d < n$).
> In questo caso $R = \mathbb{N}$, e per ogni istanza n del problema, $\rho(\eta(S(n)))$ è un qualunque elemento di $\eta(S(n))$ diverso da 1 e da n.
> **Attenzione**: il numero d potrebbe non esistere!

> **Esempio 4**
> Dato $n \in \mathbb{N}$, calcolare **il più grande** divisore d non banale di n (ossia $d > 1 \land d < n$).
> In questo caso $R = \mathbb{N}$, e per ogni istanza n del problema, $\rho(\eta(S(n)))$ è il più grande elemento di $\eta(S(n))$ diverso da 1 e da n.
> **Attenzione**: il numero d potrebbe non esistere!

Tutti questi problemi possono essere classificati in base al tipo di problema, infatti:

- **Esempio 1** è un *problema di enumerazione*, in quanto ci viene richiesto di elencare tutte le soluzioni effettive.
- **Esempio 2** è un *problema di decisione (o decisionale)*, in quanto ci viene richiesto di decidere se l'istanza possiede una certa proprietà.
- **Esempio 3** è un *problema di ricerca*, in quanto viene richiesto di trovare (e mostrare) una qualunque soluzione effettiva.
- **Esempio 4** è un *problema di ottimizzazione*, in quanto alle soluzioni effettive è associata una misura e viene richiesto di trovare una soluzione effettiva di misura massima, oppure minima.

I due diversi tipi di macchine di Turing che abbiamo visto risolvono i diversi tipi di problemi:

- le macchine di tipo **trasduttore** risolvono i problemi di ricerca, enumerazione e ottimizzazione
- le macchine di tipo **riconoscitore** risolvono i problemi di decisione
  La teoria della complessità, vista nelle lezioni precedenti, si occupa per lo più di *decidere l'appartenenza di parole ad insiemi di parole* utilizzando riconoscitori.
  D'ora in avanti ci occuperemo solo di *problemi decisionali*, per estendere quanto studiato nelle lezioni precedenti.

Problemi decisionali

Un problema è una quintupla $\langle \mathfrak{I}, S, \eta, \rho, R \rangle$ dove:

- η è il sottoinsieme di S che specifica quali fra le *soluzioni possibili*, sono le **soluzioni effettive** per una data istanza $x \in \mathfrak{I}$
- ρ è la funzione che associa all'insieme delle soluzioni effettive $\eta(S(x))$, una risposta all'istanza x del problema.
  Nel caso dei problemi decisionali $R = \{\text{vero}, \text{falso}\}$, questo significa che **ρ è un predicato**, ovvero una funzione logica il cui valore di verità dipende da qualche incognita. Possiamo quindi riassumere η, ρ, R in un unico predicato π, ovvero $\pi(x, S(x)) = \text{vero}$ **se e solo se l'insieme delle soluzioni possibili per x soddisfa i vincoli del problema**.
  Quindi un *problema decisionale* è descritto da una tripla $\langle \mathfrak{I}, S, \pi \rangle$.

> **Esempio 1**
> Dato un grafo non orientato G, una coppia di nodi s e t e un intero k, decidere se esiste in G un percorso da s a t di lunghezza k.
>
> - $\mathfrak{I} = \{\langle G, s, t, k \rangle : G \text{ è un grafo non orientato} \wedge s, t \text{ sono due nodi di } G \wedge k \in \mathbb{N}\}$
> - $S(G, s, t, k) = \{\langle u_0, \dots, u_k \rangle : \text{per } i = 0, \dots, k,\ u_i \text{ è un nodo del grafo}\}$
> - $\pi(G, s, t, k, S(G, s, t, k)) = \exists \langle u_0, \dots, u_k \rangle \in S(G, s, t, k) : s = u_0 \wedge t = u_k \wedge \forall i = 0, \dots, k,$

> **Esempio 2**
> Dato un insieme X di variabili booleane ed un predicato f, definito sulle variabili X e contenente i soli operatori $\wedge, \vee, \neg$, decidere se esiste una assegnazione a di valori in $\{\text{vero}, \text{falso}\}$ alle variabili X tali che $f(a(X)) = \text{vero}$
>
> - $\mathfrak{I} = \{\langle X, f \rangle : X \text{ è un insieme di variabili booleane} \wedge f \text{ è un predicato su } X\}$
> - $S(X, f) = \Big\{a : X \to \{\text{vero}, \text{falso}\}\Big\}$ (S è l'insieme delle assegnazioni di verità alle variabili in X)
> - $\pi(X, f, S(X, f)) = \exists a \in S(X, f) : f(a(X)) = \text{vero}$

Formalizzato il concetto di problema decisionale, siamo quasi pronti ad estendere quanto studiato sulla complessità dei linguaggi alla complessità dei problemi decisionali, e visto che la

complessità dei linguaggi è studiata utilizzando la macchina di Turing, utilizzeremo quest'ultima anche per studiare la complessità dei problemi decisionali.
Ma per utilizzare una macchina di Turing per *decidere* un problema decisionale abbiamo bisogno di trasformare le istanze di quel problema in parole e quindi occorre **codificare** le istanze di un problema decisionale.

Codifica

In questo paragrafo cerchiamo di capire il concetto di codifica di un problema decisionale.
Iniziamo con l'esempio 2, che è un problema di **soddisfacibilità** (in breve **SAT**). Consideriamo un caso particolare di questo problema, il **3SAT**.

Una funzione booleana f si dice in *forma normale congiuntiva* se f è la congiunzione di un certo numero $m \in \mathbb{N}$ di clausole, ossia, $f = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ dove ogni clausola c_j è la disgiunzione di variabili di X. Una funzione in forma normale congiuntiva è la seguente

$$f = \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3)}_{c_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{c_2}$$

Nel problema **3SAT** le istanze sono funzioni booleane le cui clausole sono costituite da esattamente 3 variabili.
Mostriamo ora due diverse codifiche per $\mathfrak{I}_{3\mathrm{SAT}}$.

Prima codifica per il 3SAT

Chiamiamo questa codifica χ_1. Rappresentiamo ciascuna variabile di $X = \{x_1, x_2, \ldots, x_n\}$ con $n = |X|$ bit. Ricordiamo che la variabile x_i è codificata dalla parola di n caratteri il cui unico 1 è quello in posizione i. Codifichiamo poi ogni letterale x_i in una clausola c_j, con la sua codifica, preceduta da uno 0 se il letterale è non negato, mentre è preceduta da un 1 se il letterale è negato. Gli $\vee$ in una clausola sono codificati col numero 2, mentre gli $\wedge$ sono rappresentati dal numero 3. Premettiamo alla codifica di f tanti 4 quanti gli elementi di X.

Esempio
Se $X = \{x_1, x_2, x_3\}$ e $f = c_1 \wedge c_2$ con $c_1 = x_1 \vee x_2 \vee x_3$ e $c_2 = x_1 \vee \neg x_2 \vee \neg x_3$ rappresentiamo f nel seguente modo

$$f = 444\, \underbrace{0100}_{x_1}\, 2\, 0010\, \underbrace{\;2\;}_{\vee}\, 0001\, \underbrace{\;3\;}_{\wedge}\, 0100\, 2\, \underbrace{1010}_{\neg x_2}\, 2\, 1001$$

Seconda codifica per il 3SAT

Chiamiamo questa codifica χ_2. Codifichiamo f in forma esplicita. Qualunque funzione è descritta completamente descrivendo i valori che assume in **tutti** i punti del suo insieme di esistenza.
Naturalmente, se una funzione è definita su $\mathbb{N}$ non possiamo descrivere il valore che essa assume per ogni $n \in \mathbb{N}$, perché i numeri naturali sono infiniti.
La nostra f dell'istanza $\langle X, f \rangle$ di **3SAT** è definita su $\{\text{vero}, \text{falso}\}^{|X|}$ e poiché X è un insieme finito, l'insieme di esistenza di f è finito. Possiamo quindi codificare f in forma esplicita mediante la sua **tavola di verità**.

> **Esempio**
> Se $X = \{x_1, x_2, x_3\}$ e $f = c_1 \wedge c_2$ con $c_1 = x_1 \vee x_2 \vee x_3$ e $c_2 = x_1 \vee \neg x_2 \vee \neg x_3$ rappresentiamo f nel seguente modo
>
> “FI/MOD II/img/img14.png” non trovato.
>
> Codificando "vero" con 1 e "falso" con 0 e scrivendo le righe della tabella una di seguito l'altra separate da 2, quindi la tabella è codificata nel seguente modo
>
> $$1111\ 2\ 1101\ 2\ 1011\ 2\ 1001\ 2\ 0110\ 2\ 0101\ 2\ 0011\ 2\ 0000$$

Consideriamo il seguente algoritmo che verifica, dato $\langle f, X \rangle \in I_{3\text{SAT}}$ se f è *soddisfacibile*, ossia, se esiste un assegnazione a di valori in $\{\text{vero}, \text{falso}\}$ alle variabili di X tali che $f(a(X)) = \text{vero}$:

1. calcola $n = |X|$
2. per ogni assegnazione di verità a alle variabili di X, verifica se $f(a(X)) = \text{vero}$ e in tal caso termina nello stato di accettazione q_A
3. se non ha mai terminato in q_A nel passo 2, termina nello stato di rigetto q_R
   Vediamo ora quest'algoritmo in esecuzione su un istanza di **3SAT** utilizzando entrambe le codifiche.

Se $\langle X, f \rangle$ è codificata secondo la codifica χ_1 utilizziamo una mdT T_1 a due nastri e che opera in due fasi:

- all'inizio della computazione $\chi_1(X, f)$ è scritta sul primo nastro, mentre il secondo nastro è vuoto
- Fase 1: utilizzando i 4 iniziali della codifica di $\langle X, f \rangle$, scrive sul secondo nastro tutte le parole binarie di lunghezza $|X|$, separate l'una dall'altra con un 5: ciascuna parola binaria corrisponde ad una assegnazione di verità agli elementi di X, ad esempio, se $|X| = 3, 010$ corrisponde a $a(x_1) = \text{falso}, a(x_2) = \text{vero}, a(x_3) = \text{falso}$

- Fase 2: per ogni assegnazione di verità a scritta sul secondo nastro, utilizzando la codifica di f scritta sul primo nastro, verifica se a soddisfa f. Se ciò accade, accetta e termina.
- Se la fase 2 è terminata senza accettare, allora rigetta.
  Quanto vale $\mathrm{dtime}(T_1, \chi_1(X, f))$?
- Fase 1: se $n = |X|$, la fase 1 richiede almeno 2^n passi (lo stesso numero di assegnazioni possibili)
- $|\chi_1(X, f)| < n + [3(n+1)+3](2n)^3 < n^4 + 7n(8n^3) < 57n^4$
  e quindi

  $$\mathrm{dtime}(T_1, \chi_1(X, f)) > 2^n > 2^{\frac{\sqrt[4]{\chi_1(X,f)}}{57}}$$

  Se $\langle X, f\rangle$ è codificata secondo la codifica χ_2, ad esempio

  $$1111\ 2\ 1101\ 2\ 1011\ 2\ 1001\ 2\ 0110\ 2\ 0101\ 2\ 0011\ 2\ 0000$$

  utilizziamo una mdT T_2 ad un solo nastro. All'inizio della computazione $\chi_2(X, f)$ è scritta sul nastro. T_2 scandisce l'input: poiché il carattere a sinistra di un 2 è il valore assunto da f quando alle sue variabili sono assegnati i valori a sinistra di quel carattere, se trova un 1 a sinistra di un 2, allora accetta e termina. Poiché $\chi_2(X, f)$ contiene in sé tutte le possibili assegnazioni di verità alle variabili in f, se T_2 ha terminato la scansione dell'input senza accettare, allora rigetta.
  Quanto vale $\mathrm{dtime}(T_2, \chi_2(X, f))$?
  Questa volta è molto facile, perché T_2 deve scandire l'input una sola volta, quindi
  $\mathrm{dtime}(T_2, \chi_2(X, f)) = |\chi_2(X, f)|$

Riassumendo

- se $\langle X, f\rangle$ è codificata secondo χ_1 implementiamo l'algoritmo mediante una macchina T_1 tale che $\mathrm{dtime}(T_1, \chi_1(X, f)) > 2^{\beta(n)}$ dove $\beta(n) = \sqrt[4]{|\chi_1(X, f)|}$, quindi l'algoritmo che decide **3SAT** impiega *tempo esponenziale* nella lunghezza di χ_1.
- se $\langle X, f\rangle$ è codificata secondo χ_2, implementiamo l'algoritmo mediante una macchina T_2 tale che $\mathrm{dtime}(T_2, \chi_2(X, f)) = |\chi_2(X, f)|$, ossia lo stesso algoritmo che decide **3SAT** impiega *tempo lineare* nella lunghezza di χ_2

Ora, ricordando che un linguaggio è nella classe $\mathbf{P}$ se esiste una mdT deterministica che lo decide in tempo polinomiale, possiamo concludere che il linguaggio associato a **3SAT** appartiene a $\mathbf{P}$?

Oss.
T_1 e T_2 implementano lo stesso algoritmo, ma operano su due codifiche diverse

Dunque la caratteristica *"essere un algoritmo polinomiale"* dipende dal modo in cui è codificato l'input? Diciamo sì e no.
Poiché la complessità di un algoritmo è espressa in termini di lunghezza dell'input, e quindi da come viene codificato, e siccome noi la codifica possiamo renderla lunga quanto vogliamo, aggiungendo ad esempio un sacco di caratteri senza senso, possiamo ad esempio prendere la codifica χ_1 e aggiungere alla fine $2^{|X|}$ caratteri 5 ottenendo

$$\chi_3(X, f) = 444\ 0100\ 2\ 0010\ 2\ 0001\ 3\ 0100\ 2\ 1010\ 2\ 1001\ 55555555$$

in questo modo $|\chi_3(X, f)| > 2^n$ e da questa codifica deriveremmo una macchina T_3 per **3SAT** tale che $\text{dtime}(T_3, \chi_3(X, f)) \in O(|\chi_3(X, f)|)$, ma questa codifica è **irragionevolmente** lunga.

Ripensiamo alle codifiche χ_1 e χ_2:

- la codifica di χ_1 rappresenta di $\langle X, f\rangle$ solo l'informazione *strettamente necessaria*, ossia la **struttura di f**
- la codifica di χ_2 rappresenta, invece, $\langle X, f\rangle$ in forma estesa, infatti χ_2 contiene la soluzione del problema così che, per trovare la soluzione è sufficiente leggere la codifica, ma questo significa che calcolare la codifica di χ_2 ha richiesto molto tempo, ossia, **il tempo impiegato dalla computazione $T_1(\chi_1(X, f))$ lo dobbiamo impiegare noi per calcolare $\chi_2(X, f)$ se vogliamo utilizzare questa codifica**. In effetti χ_2 è *esponenzialmente* più lunga di χ_1

Informalmente, **una codifica χ per un problema Γ è irragionevole se esiste un'altra codifica $\chi^{'}$** tale che le parole in cui χ codifica le istanze di Γ sono "più che polinomialmente" lunghe delle parole in cui $\chi^{'}$ codifica le istanze di Γ.

Questo significa che esiste una funzione più che polinomiale f tale che, per qualche istanza x di Γ $|\chi(X)| \geq f(|\chi^{'}|)$, quindi $f : \mathbb{N} \to \mathbb{N}$ è più che polinomiale se, per ogni $k \in \mathbb{N}$, $f(n) \in \Omega(n^k)$.
Informalmente, il rapporto fra $|\chi(X)|$ e $|\chi^{'}(X)|$ è più grande di qualsiasi polinomio.
Quello che accadeva tra χ_1 e χ_2 era proprio che χ_2 è una **codifica irragionevole** di **3SAT**.

Quindi, **una codifica χ per un problema Γ è ragionevole se:

- comunque si scelga un'altra codifica $\chi^{'}$ per Γ, esistono tre interi k, h_1 e h_2 tali che, per ogni istanza x di Γ, $|\chi(x)| \leq h_1|\chi^{'}(x)|^k + h_2$.
  Questo significa che, se χ è una codifica ragionevole per Γ, comunque scegliamo un'altra codifica $\chi^{'}$ per Γ, può succedere che le parole risultanti dalla codifica $\chi^{'}$ siano più corte delle parole risultanti dalla codifica χ, ma esiste un polinomio p tale che, qualunque sia l'istanza x di Γ, $|\chi(x)|$ non è più grande di $p(|\chi^{'}(x)|)$.

Alla luce di quanto detto fino ad ora, dovrebbe essere chiaro che possiamo estendere ai problemi tutto quello studiato relativamente alla complessità di linguaggi, a patto di **utilizzare codifiche**

ragionevoli per codificare le istanze dei problemi, perché quando si utilizzano codifiche irragionevoli non ha più senso parlare della complessità di un problema, perché potremmo aver trasferito nella complessità della codifica la complessità di risoluzione del problema, esattamente come fatto nel caso della codifica χ_2 del problema **3SAT**. Per questo d'ora in poi, faremo riferimento sempre a codifiche ragionevoli.

Lezione 17 - Complessità di problemi

Nella lezione precedente abbiamo quindi proposto di estendere ai problemi quanto studiato relativamente alla complessità dei linguaggi, a patto di utilizzare *codifiche ragionevoli* per codificare le istanze dei problemi.
Dobbiamo solo capire come trasformare un problema in un linguaggio e se questo è semplice o ci porterà ad affrontare nuove questioni.

Sia $\Gamma = \langle \mathfrak{I}_\Gamma, S_\Gamma, \pi_\Gamma \rangle$ un problema decisionale. Osserviamo che l'insieme delle istanze $\mathfrak{I}_\Gamma$ è partizionato in due sottoinsiemi:

- l'insieme delle **istanze positive** ovvero quelle che verificano π_Γ
- l'insieme delle **istanze negative** ovvero quelle che non verificano π_Γ
  Sia $\chi : \mathfrak{I}_\Gamma \to \Sigma^*$ una codifica ragionevole per Γ.
  La codifica χ partiziona Σ^* in tre sottoinsiemi di parole:
- l'insieme Y_Γ delle parole che codificano istanze positive di Γ
- l'insieme N_Γ delle parole che codificano istanze negative di Γ
- l'insieme delle parole che *non* codificano istanze di Γ
  Il linguaggio associato a Γ mediante la codifica χ è il sottoinsieme $L_\Gamma(\chi)$ di Σ^* contenenti le parole appartenenti a Y_Γ, ossia

$$L_\Gamma(\chi) = \left\{ x \in \Sigma^* : \exists y \in \mathfrak{I}_\Gamma \left[x = \chi(y) \wedge \pi_\Gamma(y, S_\Gamma(y)) \right] \right\}$$

  Quindi, decidere se un'istanza y di Γ è un'istanza positiva, corrisponde a decidere se $x = \chi(y)$ è contenuto in $L_\Gamma(\chi)$ e quindi data $x \in \Sigma^*$, per decidere se $x \in L_\Gamma(\chi)$ occorre:
- decidere se x è la codifica di un'istanza y di Γ
- in caso affermativo, decidere se il predicato $\pi_\Gamma(y, S_\Gamma(y))$ è soddisfatto

Possiamo quindi definire la complessità computazionale di un problema decisionale.

Def.
Sia $\Gamma = \langle \mathfrak{I}_\Gamma, S_\Gamma, \pi_\Gamma \rangle$ un problema decisionale e sia C una classe di complessità.

- Data una funzione f totale e calcolabile
- e $C \in \left\{ \text{DTIME}[f(n)], \text{DSPACE}[f(n)], \text{NTIME}[f(n)], \text{NSPACE}[f(n)] \right\}$
  Diciamo che $\Gamma \in C$ se esiste una codifica *ragionevole* $\chi : \mathfrak{I}_\Gamma \to \Sigma^*$ per Γ tale che

$L_\Gamma(\chi) \in C$

Vediamo con un esempio cosa occorre fare per decidere se $x \in L_\Gamma(\chi)$.

> **Esempio**
> Riprendiamo il problema **3SAT** e la codifica χ_1, che abbiamo visto essere una codifica ragionevole
>
> $$\chi_1(X, f) = 444\ 0100\ 2\ 0010\ 2\ 0001\ 3\ 0100\ 2\ 1010\ 2\ 1001$$
>
> Allora una parola $x \in \{0, 1, 2, 3, 4\}^*$ è in $L_{3SAT}(\chi_1)$ se sono verificati i seguenti punti:
>
> - x deve essere la codifica secondo χ_1 di qualche coppia $\langle X, f\rangle$ istanza di **3SAT**, infatti è semplice verificare che 40211011103420111 non è la codifica di nessuna istanza. Se x non è una codifica valida possiamo subito concludere che $x \notin L_{3SAT}(\chi_1)$
> - se x è la codifica secondo χ_1 di un'istanza $\langle X, f\rangle$ di **3SAT**, affinché $x \in L_{3SAT}(\chi_1)$ occorre che f sia soddisfacibile

Dati un problema Γ e una sua codifica ragionevole χ, per verificare che una parola sia in $L_\Gamma(\chi)$, occorre prima di tutto verificare che essa sia la codifica di una istanza.
Definiamo **il linguaggio delle istanze di Γ**, ossia, il linguaggio

$$\chi(\mathfrak{I}_\Gamma) = \Big\{x \in \Sigma^* : \exists y \in \mathfrak{I}_\Gamma[x = \chi(y)]\Big\}$$

e osserviamo che χ è una codifica di $\mathfrak{I}_\Gamma$, quindi, se $y, z \in \mathfrak{I}_\Gamma$ sono due istanze di Γ con $y \neq z$, allora $\chi(y) \neq \chi(z)$ quindi χ è una funzione invertibile, allora possiamo definire il linguaggio $L_\Gamma(\chi)$ anche nella maniera seguente:

$$L_\Gamma(\chi) = \Big\{x \in \Sigma^* : x \in \chi(\mathfrak{I}_\Gamma) \wedge \pi\big(\chi^{-1}(x)),\ S_\Gamma(\chi^{-1}(x)\big)\Big\}$$

Dunque, se, per decidere se una parola x appartiene a $L_\Gamma(\chi)$ dobbiamo anche verificare se x è effettivamente la codifica di un'istanza di Γ, allora per definire la complessità del problema decisionale Γ occorre considerare anche la complessità di decidere il linguaggio $\chi(\mathfrak{I}_\Gamma)$.

Esempio
Consideriamo il problema decisionale del *Percorso in Ciclo Hamiltoniano (PHC)*.
Sia dato un particolare grafo non orientato $G = (V, E)$, che contiene un ciclo che passa una ed una sola volta per ciascuno dei suoi nodi. Siano dati due nodi $u, v \in V$. Si chiede di decidere se esiste in G un percorso che collega u a v.

Formalizziamo il problema precedente mediante la tripla $\langle \mathfrak{I}_{PHC}, S_{PHC}, \pi_{PHC} \rangle$:

- $\mathfrak{I}_{PHC} = \{\langle G = (V, E), u, v \rangle\}$, G è un grafo non orientato $\wedge$ $\exists$ un ciclo c in G che passa una e una sola volta attraverso ciascun nodo di $G \wedge u, v \in V$
- $S_{PHC}(G, u, v) = \{p : p \text{ è un percorso in } G\}$
- $\pi_{PHC}(G, u, v, S_{PHC}(G, u, v)) = \exists\, p \in S_{PHC}(G, u, v)$ che connette u a v
  Se sappiamo che un grafo contiene un ciclo che passa una e una sola volta attraverso tutti i nodi di G, allora, qualunque coppia di nodi u e v si consideri, una porzione di quel ciclo è un percorso da u a v.
  Questo significa che ogni istanza del problema PHC è una istanza sì, quindi indipendentemente dalla codifica utilizzata, decidere se una qualunque istanza del problema soddisfa il predicato del problema richiede costo costante.
  D'altra parte, data una qualunque codifica ragionevole χ per PHC, per decidere se una parola $x \in \{0,1\}^*$ è contenuta in $L_{PHC}(\chi)$ dobbiamo verificare
- sia se x è la codifica di una istanza di PHC, ossia di un grafo che contiene un ciclo che attraversa tutti i nodi una e una sola volta e di una coppia di suoi nodi
- sia se il grafo contiene un percorso che connette i due nodi
  La prima di queste due verifiche, ossia decidere $\chi(\mathfrak{I}_{PHC})$ è un linguaggio $\mathbf{NP}$-completo e quindi concludiamo che $L_{PHC}(\chi)$ è $\mathbf{NP}$-completo.
  Allora, anche se, una volta assodato che una parola $x \in \{0,1\}^*$ è istanza di PHC, decidere se x soddisfa $\pi_{PHC}(G, u, v, S(G, u, v))$ ha costo costante, **non possiamo affermare che decidere PHC è un problema in $\mathbf{P}$**.

Sia Σ un qualunque alfabeto. Una qualunque codifica χ delle istanze di un problemadecisionale Γ in parole di Σ^* induce una tri-partizione di Σ^*, ovvero una partizione di Σ^* in tre sottoinsiemi:

- l'insieme Y_Γ delle parole di Σ^* che codificano le istanze sì di Γ, ovvero il linguaggio $L_\Gamma(\chi)$
- l'insieme N_Γ delle parole di Σ^* che codificano le istanze no di Γ
- parole di Σ^* che non codificano istanze di Γ
  Ricordiamo che, dato un qualunque linguaggio $L \subseteq \Sigma^*$, il linguaggio complemento di L è $L^C = \Sigma^* - L$.
  Quindi secondo questa definizione il linguaggio complemento di $L_\Gamma(\chi)$ è

  $$L_\Gamma(\chi)^C = \Sigma^* - L_\Gamma(\chi)$$

  ovvero tutte le parole di Σ^* che codificano istanze no di Γ e tutte le parole di Σ^* che non codificano istanze di Γ.
  Ma siamo sicuri che questo è proprio ciò che corrisponde al complemento di un problema decisionale?
  Se pensiamo al complemento di un problema di decisione, quello che ci viene in mente sono **le istanze del problema che non soddisfano il predicato**, ad esempio, il problema

$3SAT^C$ è l'insieme delle istanze $\langle X, f\rangle$ di **3SAT** tali che f non è soddisfacibile, quindi

$$3SAT^C = \langle \mathfrak{I}_{3SAT}, S_{3SAT}, \neg\pi_{3SAT}\rangle$$

Perciò il linguaggio che vogliamo associare al problema complemento di Γ non è $L_\Gamma(\chi)^C = \Sigma^* - L_\Gamma(\chi)$, bensì l'insieme $N\Gamma$

$$L_{\Gamma^C}(\chi) = \{x \in \Sigma^* : x \in \chi(\mathfrak{I}_\Gamma) \wedge \neg\pi_\Gamma(\chi^{-1}(x), S_\Gamma(\chi^{-1}(x)))\}$$

Ora, datu un linguaggio L e una classe di complessità C, sappiamo, per definizione, che se $L_\Gamma(\chi) \in C$ allora $(L_\Gamma(\chi))^C \in coC$. Ma se sappiamo che $L_\Gamma(\chi) \in C$, cosa possiamo dire del suo linguaggio complementare $L_{\Gamma^C}(\chi)$? Ovvero, se sappiamo classificare un problema di decisione, sappiamo classificare anche il suo complemento? Vediamo un esempio e poi rispondiamo a questa questione.

> **Esempio**
> Riprendiamo il prolema decisionale PHC. Dato un grafo non orientato $G = (V, E)$ che contiene un ciclo che passa una ed una sola volta per ciascuno dei suoi nodi, e dati due suoi nodi $u, v \in V$, si chiede di decidere se esiste in G un percorso che collega u a v.
> PHC è formalizzato mediante la tripla $\langle \mathfrak{I}_{PHC}, S_{PHC}, \pi_{PHC}\rangle$:
>
> - $\mathfrak{I}_{PHC} = \{\langle G = (V, E), u, v\rangle\}$, G è un grafo non orientato $\wedge$ $\exists$ un ciclo c in G che passa una e una sola volta attraverso ciascun nodo di $G \wedge u, v \in V$
> - $S_{PHC}(G, u, v) = \{p : p \text{ è un percorso in } G\}$
> - $\pi_{PHC}(G, u, v, S_{PHC}(G, u, v)) = \exists\, p \in S_{PHC}(G, u, v)$ che connette u a v
> PHC^C allora è:
>   dato un grafo non orientato $G = (V, E)$ che contiene un ciclo che passa una ed una sola volta per ciascuno dei suoi nodi, e dati due suoi nodi $u, v \in V$, si chiede di decidere se **non esiste** in G alcun percorso che collega u a v ed è formalizzato mediante la tripla $\langle \mathfrak{I}_{PHC}, S_{PHC}, \neg\pi_{PHC}\rangle$ con
>
> $$\neg\pi_{PHC}(G, u, v, S_{PHC}(G, u, v)) = \nexists\, p \in S_{PHC}(G, u, v) \text{ che connette u a v}$$
>
>   Data una qualunque codifica ragionevole χ per PHC^C, per decidere se una parola x è contenuta in $L_{PHC^C}(\chi)$, dobbiamo verificare:
> - se x è la codifica di un'istanza di PHC^C, ossia di un grafo che contiene un ciclo che attraversa tutti i nodi una e una sola volta, e di una coppia di suoi nodi
> - se questo grafo non contiene percorsi che connettono i due nodi
>   Come abbiamo visto:

> - la verifica che x sia la codifica di un'istanza di PHC è un problema $\mathbf{NP}$-completo
> - verificare se una qualunque istanza del problema soddisfa il predicato del problema richiede costo costante, proprio perché nessuna istanza soddisfa il predicato!
>   Possiamo quindi concludere (ad occhio) che PHC^C è $\mathbf{NP}$-completo.

Riassumendo, il problema PHC è $\mathbf{NP}$-completo e anche il suo complemento PHC^C.
Quindi sembrerebbe che **non** possiamo trasportare ai problemi decisionali la teoria della complessità che abbiamo sviluppato per i linguaggi, perché **la complessita di un problema decisionale dipende anche dalla complessità di decidere il linguaggio delle istanze**. Ma se la decisione del linguaggio delle istanze richiede "poche risorse", possiamo trasferire tutto ciò che abbiamo studiato relativamente alla complessità dei linguaggi, alla complessità dei problemi decisionali, come ci mostra il prossimo teorema.

Teorema 7.1
Sia $\Gamma = \langle \mathfrak{I}_\Gamma, S_\Gamma, \pi_\Gamma \rangle$ un problema decisionale e sia $\chi : \mathfrak{I}_\Gamma \to \Sigma^*$ una sua codifica ragionevole.
Se $\chi(\mathfrak{I}_\Gamma) \in \mathbf{P}$ allora valgono le seguenti implicazioni:

- se $L_\Gamma(\chi) \in \mathbf{NP}$ allora $L_{\Gamma^C}(\chi) \in co\mathbf{NP}$
- se $L_\Gamma(\chi) \in \mathbf{NEXPTIME}$ allora $L_{\Gamma^C}(\chi) \in co\mathbf{NEXPTIME}$

Dim. per il caso 1.
Se $\chi(\mathfrak{I}_\Gamma) \in \mathbf{P}$, allora esistono una macchina deterministica T ed un intero h tali che, per ogni $x \in \Sigma^*$, T decide se $x \in \chi(\mathfrak{I}_\Gamma)$ e $\text{dtime}(T, x) \in O(|x|^h)$.
Se $L_T(\chi) \in \mathbf{NP}$, allora esistono una macchia non deterministica NT ed un intero k tali che, per ogni $x \in L_\Gamma(\chi)$, NT accetta x e $\text{ntime}(NT, x) \in O(|x|^k)$.
Combinando T e NT, costruiamo una nuova macchina non determinstica NT_0 che **accetta il linguaggio complemento di $L_{\Gamma^C}(\chi)$, ossia, che accetta $(L_{\Gamma^C}(\chi))^C$**.
Sorgono due domande:

1. perché ci interessa accettare $(L_{\Gamma^C}(\chi))^C$? Se riusciamo a mostrare che possiamo accettare $(L_{\Gamma^C}(\chi))^C$ in tempo non deterministico polinomiale, allora $(L_{\Gamma^C}(\chi))^C$ è in $\mathbf{NP}$ e dunque $L_{\Gamma^C}(\chi) \in co\mathbf{NP}$
2. che parole troviamo in $(L_{\Gamma^C}(\chi))^C$? Poiché in $L_{\Gamma^C}(\chi)$ troviamo parole che codificano le istanze no di Γ, allora in $(L_{\Gamma^C}(\chi))^C$ troviamo:
   - parole che non codificano istanze di Γ
   - parole che codificano istanze sì di Γ, ossia, parola che appartengono a $L_\Gamma(\chi)$
   NT_0 opera in due fasi: con input $x \in \Sigma^*$:
3. Simula la computazione $T(x)$, se $T(x)$ termina nello stato di rigetto, allora NT_0 termina nello stato di accettazione, altrimenti inizia la fase 2
4. Simula la computazione $NT(x)$, se $NT(x)$ accetta, allora NT_0 accetta
   $NT_0(x)$ accetta quando $x \notin \chi(\mathfrak{I}_\Gamma)$ oppure quando $x \in L_\Gamma(\chi)$, cioè, **$NT_0(\chi)$ accetta se e soltanto se x appartiene a $(L_{\Gamma^C}(\chi))^C$**.

È semplice verificare che $\mathrm{ntime}(NT_0, x) \in O(|x|^{\max\{h,k\}})$.
Quindi $(L_{\Gamma^C}(\chi))^C$ è in $\mathbf{NP}$ e dunque $L_{\Gamma^C}(\chi) \in co\mathbf{NP}$. $\square$

Non è ragionevole che sia più complesso decidere se una parola è istanza di un problema, che decidere se una istanza di quel problema è un'istanza positiva, perché la difficoltà nel risolvere un problema non dovrebbe essere nel riconoscere che i dati che ci vengono forniti siano effettivamente dati del nostro problema, ma nel trovare una soluzione ad una data istanza del problema.
Per questo, d'ora in poi assumeremo che, per ogni problema di decisione Γ e per ogni sua codifica ragionevole χ, il linguaggio delle istanze sia in $\mathbf{P}$, ossia $\chi(\mathfrak{I}_\Gamma) \in \mathrm{P}$.
Questo significa che, la formalizzazione del problema PHC sarà:

- $\mathfrak{I}_{PHC} = \{\langle G = (V, E), u, v \rangle\}$, G è un grafo non orientato $\wedge$ $u, v \in V$
- $S_{PHC}(G, u, v) = \{p : p \text{ è un percorso in } G\}$
- $\pi_{PHC}(G, u, v, S_{PHC}(G, u, v)) = \exists$ un ciclo c in G che passa una e una sola volta attraverso ciascun nodo di $G \wedge \exists\, p \in S_{PHC}(G, u, v)$ che connette u a v
  ossia, sposteremo nel predicato tutte le proprietà che devono essere soddisfatte dai dati che costituiscono l'istanza.

Lezione 18 - Classe NP

La classe NP

Prima di addentrarci in questioni strutturali, cerchiamo di capire, perchè la classe NP è così importante?
Tanto importante che qualcuno ha pensato di mettere una taglia da un milione di dollari sulla congettura $P \neq NP$!!
La classe P è importante perchè se collochiamo un problema in P quel problema sappiamo risolverlo per davvero.
MA, La classe NP?
Cosa ci importa di sapere se un certo problema per il quale magari non riusciamo a trovare un algoritmo polinomiale $\in NP$.
Che ce ne importa di sapere che quel problema è deciso da una macchina
non deterministica in tempo polinomiale?

L'importanza della classe NP

Se l'importanza di NP non va individuata nel modello di calcolo sul quale è basata, allora non può che risiedere nei problemi che la popolano!
In effetti nella classe NP si trovano tanti problemi:

- Acquistare i biglietti aerei per un giro di tutte le capitali dell'UE, spendendo in totale al massimo 10000 euro
- Suddividere un insieme di oggetti (ciascuno di peso diverso) sui due piatti di una bilancia in modo tale che alla fine la bilancia sia in equilibrio
- Piastrellare un pavimento con mattonelle di forme e dimensioni diverse in modo tale che non rimangano spazi scoperti
- Scegliere al più 10 rappresentanti degli studenti ai quali comunicare una direttiva in modo tale che ogni altro studente conosca almeno uno di quelli che sono stati scelti così da poter essere informato
- ...e tanti (ma tanti) altri...
  Che hanno grande rilevanza pratica
  Che non si riesce a risolvere mediante algoritmi (deterministici) polinomiali ...

Ma che *non si riesce* nemmeno a dimostrare che non possono essere risolti in tempo deterministico polinomiale

La struttura dei problemi in NP

Dunque, sappiamo che i NP si trovano molti problemi (decisionali) importanti, e sappiamo anche che un problema é in NP quando esiste una macchina di turing non deterministica che **accetta** le sue istanze **si** in tempo polinomiale.
Ma allora perché continuiamo a dire che in NP si trovano i linguaggi **accettati** in tempo non deterministico polinomiale? perche continuiamo a non usare la parola "**decisi**"?
Per comprendere questo particolare dobbiamo tornare indietro di qualche lezione: il **Genio burlone e pasticcione** che costituisce uno dei modelli di una computazione non deterministica.

> Quando durante una computazione non deterministica $NT(x)$, la macchina si trova in un certo stato q e legge un certo simbolo s, e nell'insieme delle quintuple NT essitono tante quintuple che iniziano con la coppia (q, s), quale quintupla esegue NT?!
> (*Multiquintupla*)

In questo caso il **Genio burlone e pasticcione** ha la risposta

Multi-quintuple e Genio

Tornando a prima quindi, quale quintupla esegue tra quelle a disposizione NT?
Dato che NT é ricorsa al **genio** per decidere si aspetta che lui tramite le sue doti magiche riesca a decidere quale sia la **quintupla giusta** da eseguire
Ovvero la quintupla che, **nell'ipotesi che x sia una istanza sí del problema**, porti NT ad accettare
L'intervento del genio possiamo modellarlo tramite un'apposita istruzione in *PascalMinimo* (istruzione:*scegli*)

Istruzione scegli

“FI/MOD II/img/img15.png” non trovato.

Comprensione:

- Fino a quando la macchina NT non entra nello stato q_A o nello stato q_R
  - (e lo stato in cui si trova NT é memorizzato nella variabile q)
- Calcola l'insieme Ψ delle quintuple che puó eseguire trovandosi nello stato q e leggendo $N[T]$
  - T indica la posizione della testina sul nastro NT

- Se Ψ contiene almeno una quintupla, ne sceglie una de eseguire e la esegue
  - Gestendo le porzioni iniziali e finali del nastro mediante primaCella e ultimaCella

In questo caso, invece di simulare tutte le computazioni deterministiche di $NT(x)$, l'algoritmo si affida al **Genio** per scegliere, di volta in volta, le quintuple da eseguire.

Oss.
Se ad un certo istante Ψ non contiene quintuple e q non é q_A e non é q_R, l'algoritmo entra in loop!
Ma questo caso accade solo quando P non é totale

Ma il Genio é pasticcione

Quali conseguenze comporta ricorrere al **Genio**?
Del quale ovviamente non ci si puó fidare!
Eseguiamo l'algoritmo della Lezione 7 - Tesi di Church-Turing e Turing-equivalenza con input x - chiamiamo $\mathcal{A}$ l'algoritmo e $\mathcal{A}(x)$ la sua esecuzione sull'input x

- $\mathcal{A}(x)$ termina in q_A o in q_R
  - Ovviamente assumendo che P sia totale...e noi lo assumiamo!
- Se $\mathcal{A}(x)$ termina in q_A allora possiamo essere certi che il **Genio** ci ha indicato le risposte corrette
  - Perché il **Genio** ha trovato una sequenza di quintuple da eseguire che termina nello stato q_A, e quella sequenza costituisce una computazione accettante di $NT(x)$ e, dunque, *esiste una computazione accettante di $NT(x)$!*
- Se $\mathcal{A}(x)$ termina in q_A, allora, possiamo essere certi che possiamo accettare
- Ma se $\mathcal{A}(x)$ termina in q_R allora qualche dubbio ci viene...
  - Perché il genio ha trovato una sequenza di quintuple da eseguire che termina nello stato q_R, e quella sequenza costituisce una computazione di $NT(x)$ che rigetta e, dunque, *esiste una computazione di $NT(x)$ che rigetta*
  - Ma ovviamente, **questo non dimostra che tutte le computazioni di $NT(x)$ rigettano!**

Ecco, allora, dato che noi non ci fidiamo del **Genio**, possiamo solo concludere che il **Genio** non ha trovato la sequenza di quintuple che porta NT nello stato di accettazione.

> Ma non possiamo sapere se non l'ha trovata, perché una sequnza di quintuple che induce NT ad accettare non esiste o perché il ***Genio*** non é stato sufficientemente abile da trovarla!

Ecco quindi perché continuiamo a parlare di linguaggi **accettati**, piuttosto che **decisi**

La struttura dei problemi in NP

Il **Genio** a "mezzo servizio" gioca un ruolo fondamentale per comprendere la struttura dei problemi che popolano la classe NP.
E per comprendere questa struttura, facciamo un po' di esempi di problemi e di esempi di **algoritmi non deterministici che li risolvono**
E siccome ci accingiamo a progettare algoritmi che decidono problemi anziché linguaggi, lí descriveremo ad "alto livello", utilizzeremo il PascalMinimo, mettendo da parte le macchine di Turing.
Ma prima di fare ció, dobbiamo chiarire una piccola questione.

Quanto é potente questo Genio?

Se disponiamo di un **Genio**, perché non gli chiediamo direttamente "l'istanza x é un'istanza sí del mio problema?"
Innanzitutto, perché delle risposte del **Genio** non mi fido:

- Se gli chiedo di indicarmi quale quintupla eseguire ad un certo punto della computazione, poi posso verificare che mi ha indicato una quintupla che posso eseguire davvero
- Se gli chiedo di dirmi se x é un'istanza sí, poi come verificare effettivamente la risposta?
  Ma soprattutto, abbiamo **introdotto il Genio per modellare il non determinismo**
- Per questo gli chiediamo di scegliere quale quintupla eseguire a ciascun passo della computazione
- **E il numero di quintuple fra le quali scegliere é il grado di non determinismo della macchina**
- Che é *Costante*
  Detto questo, va bene trasportare il **Genio** nel mondo degli algoritmi di alto livello, a patto che gli proporremo sempre di operare fra un numero *Costante* di opzioni.

Il problema 3Sat

Dati un insieme X di variabili booleane ed un predicato f, definito sulle variabili in X e contenente i soli operatori $\wedge \ \vee \ \neg$, decidere se esiste una assegnazione a di valori in $\{Vero, Falso\}$ alle variabili in $X : f(a(X)) = Vero$
Consideriamo soltanto predicati f in forma 3-congiuntiva normale (3CNF), ossia,

- f é la congiunzione di un certo numero di clausole: $f = c_1 \wedge c_2 \ldots \wedge c_m$
- Ciascuna c_j é la disgiunzione ($\vee$) di tre letterali (3CNF), ad esempio $x_1 \vee \neg x_2 \vee x_3$
  Questo problema prende il nome di 3Sat, ed é cosí formalizzato:
- $\mathfrak{I}_{3Sat} = \{< X, f > \ : \ X$ é un insieme di variabili booleane $\wedge$ f é un predicato su X in 3CNF $\}$
- $S_{3Sat}(X, f) = \{a : X \to \{Vero, Falso\}\}$ (S insieme delle assegnazioni di veritá alle variabili X)

- $\pi_{3Sat}(X, f, S_{3Sat}(X, f)) = \exists\, a \in S_{3Sat}(X, f) : f(a(X)) = Vero$
  Possibile algoritmo non deterministico:

  "FI/MOD II/img/img16.png" non trovato.

  Questo algoritmo é logicamente suddiviso in 2 parti:
- La prima parte ha carattere prettamente non deterministico
  - Serve a scegliere una assegnazione di veritá a per le variabili in f
- La seconda parte ha carattere prettamente deterministico
  - Serve a verificare deterministicamente che l'assegnazione scelta soddisfi effettivamente f

> **Warning**
>
> Poiché il numero di possibilitá fra le quali scegliere ad ogni passo é pari a 2, si tratta effettivamente di un algoritmo non deterministico.
> Poiché l'algoritmo accetta se e solo se *esiste* una sequenza di scelte che soddisfa f, allora é un algoritmo che accetta 3Sat.

Complessitá:

- il primo *while* richiede tempo non deterministico lineare in $n = |X|$
- il secondo *while* richiede tempo deterministico lineare in $O(|X| \cdot |f|) = O(3nm) = O(nm)$

Conclusione:
l'algoritmo accetta $< X, f > \ \in \ \Im_{3Sat}$ in tempo $O(|X| \cdot |f|)$, e questo prova che 3Sat $\in \ NP$

Il problema CLIQUE

Il problema CLIQUE consiste nel decidere, dati un grafo non orientato $G = (V, E)$ ed un intero $k \in \mathbb{N}$, se G contiene un sottografo completo di almeno k nodi

- Formalmente, il problema é descritto dalla tripla
  - $\Im_{CLIQUE} = \{< G = (V, E), k >:\ G$ é un grafo non orientato $\wedge\ k \in \mathbb{N}\}$
  - $S_{CLIQUE} = (G = (V, E), k) = \{V' \subseteq V\}$ (S é l'insieme dei sottoinsiemi di V)
  -
  $\pi_{CLIQUE}(G, K, S_{CLIQUE}(G, K)) = \exists V' \in S(G, K) : (\forall u, v \in V'[(u, v) \in E]) \wedge |V'| \geq k$
  (ovvero, scelti due nodi in V', essi siano collegati da un arco)
  Possibile algoritmo non deterministico:

  "FI/MOD II/img/img17.png" non trovato.

  Questo algoritmo é logicamente suddiviso in 2 parti:

- La prima parte ha carattere prettamente non deterministico
  - Serve a scegliere un sottoinsieme V' di V
- La seconda parte ha carattere prettamente deterministico
  - Serve a verificare deterministicamente che il sottoinsieme scelto soddisfi effettivamente $\pi_{CLIQUE}(G, k, S_{CLIQUE}(G, k))$
  Questo algoritmo é esattamente come quello che accetta 3SAT.
  Dato che il numero di possibilitá fra le quali scegliere ad ogni passo é pari a 2, si tratta effettivamente, di un algoritmo non deterministico.
  Poiché l'algoritmo accetta se esiste una sequenza di scelte che soddisfa il predicato di CLIQUE $\pi_{CLIQUE}(G, k, S_{CLIQUE}) \implies$ accetta CLIQUE
  Complessitá
- il primo *while* richiede tempo non deterministico lineare in $n = |V|$
- il secondo *while* richiede tempo deterministico in $O(|V|^2(|V| + |E|))$
  Conclusione:
  l'algoritmo accetta $< G, k > \in \Im_{CLIQUE}$ in tempo $O(|V|^2|E|)$ e questo prova che CLIQUE $\in NP$

Ma allora...

I tre problemi che abbiamo visto in questa lezione hanno in comune la struttura del predicato π

- In tutti e tre i problemi π ha la forma: esiste almeno un elemento di S che soddisfa certe proprietá, che chiameremo η
  - $\pi(x, S(x)) = \exists y \in S(x) : \eta(x, y)$
- Non solo, ma anche gli algoritmi decisionali che abbiamo analizzato seguivano lo stesso schema: dato input x
  - F1(ND): sceglie una possibile soluzione $y \in S(x)$
  - F2(D): verifica che y soddisfa il predicato $\eta(x, y)$
- E ancora:
  - F1: sceglie una soluzione possibile che y, richiede tempo polinomiale $|x|$
  - F2: verifica che x e y soddisfino il predicato η , richiede tempo polinomiale in $|x|$
  Troppe conincidenze!!
  Beh, certamente, tutti i problemi decisionali che $\pi(x, S(x)) = \exists y \in S(x) : \eta(x, y)$ possono essere risolti da un algoritmop non deterministico che opera in 2 fasi:
- F1(ND): sceglie una possibile soluzione $y \in S(x)$
- F2(D): verifica che y soddisfa il predicato $\eta(x, y)$
  E tale algoritmo richiede tempo(ND) polinomiale se:
- F1: sceglie una soluzione possibile che y, richiede tempo polinomiale $|x|$
- F2: verifica che x e y soddisfino il predicato η , richiede tempo polinomiale in $|x|$
  Quindi:

possiamo dire che ogni problema il cui predicato ha la forma
$\pi(x, S(x)) = \exists y \in S(x) : \eta(x, y)$

- in cui la scelta di un elemento y di $S(x)$ richiede tempo polinomiale in $|x|$
- in cui la verifica che y soddisfi il predicato η, richiede tempo polinomiale in $|x|$

appartiene a NP

Lezione 19 - Caratterizzazione della classe NP

Abbiamo visto che tutti i problemi decisionali tali che:

- il predicato ha la forma $\pi(x, S(x)) = \text{esiste } y \in S(x)$ tale che $\eta(x, y)$
- la scelta di un elemento y di $S(x)$ richiede tempo non deterministico polinomiale in $|x|$
- la verifica che y soddisfi il predicato η richiede tempo polinomiale in $|x|$
  appartengono a $\mathbf{NP}$ e questi tre punti sono, quindi, condizioni sufficienti per poter affermare l'appartenenza di un problema ad $\mathbf{NP}$.

Esiste, però, un problema che non soddisfa i 3 punti, ma che appartiene comunque ad $\mathbf{NP}$? No, non è possibile e questo ce lo dice il prossimo teorema.

Teorema 9.1
Un linguaggio $L \subseteq \Sigma^*$ appartiene ad $\mathbf{NP}$ **se e soltanto se**:

- esistono una mdT deterministica T e due costanti $h, k \in \mathbb{N}$ tali che, per ogni $x \in \Sigma^*$

$$x \in L \iff \exists\, y_x \in \{0,1\}^* : |y_x| \le |x|^k$$
$$\wedge\, T(x, y_x)\text{accetta}$$
$$\wedge\, \text{dtime}(T, x, y_x) \in O(|x|^h)$$

Cosa vuol dire questo teorema?
Osserviamo che questo teorema è una **condizione necessaria e sufficiente** per poter dire che "L appartiene ad $\mathbf{NP}$", e siccome è una condizione necessaria e sufficiente, dobbiamo scomporlo in due parti.

Dimostrazione $\Longrightarrow$

Se partiamo dall'ipotesi che L appartiene ad $\mathbf{NP}$, il teorema ci indica una condizione necessaria e sufficiente per poter affermare che $x \in L$, ovvero

$$x \in L \iff \exists\, y_x \in \{0,1\}^* : |y_x| \leq |x|^k$$
$$\land\, T(x, y_x)\text{accetta}$$
$$\land\, \text{dtime}(T, x, y_x) \in O(|x|^h)$$

Ma cosa vuol dire questa condizione?
Per poter affermare che $x \in L$ allora:

- $\exists\, y_x \in \{0,1\}^*$ ci dice che dobbiamo trovare una parola y da associare ad x
- $|y_x| \leq |x|^k$ che non sia troppo lunga
- $\land\, T(x, y_x)$ accetta e che induca *una certa macchina deterministica* T ad accettare
- $\land\, \text{dtime}(T, x, y_x) \in O(|x|^h)$ e ad accettare in tempi brevi!

Il teorema quindi ci dice che se $L \in \mathbf{NP}$ allora esiste una mdT deterministica T tale che, se le do in input due parole x ed y, con y scelta da me e non troppo lunga, la computazione $T(x, y)$, in tempo polinomiale in $|x|$, *accetta se e solo se* $x \in L$ ed ho scelto la y giusta. Infatti il teorema dice che **riesco a trovare una parola y_x che possa convincere T ad accettare se $x \in L$, ma non se $x \notin L$**!

Quindi se trovassi qualcuno in grado di suggeririmi, per ogni $x \in L$, la parola y_x giusta, allora riesco ad accettare le parole di L in tempo deterministico polinomiale.
Facciamo quindi tornare il nostro genio, ma invece di chiedergli una quintupla per volta, gli chiediamo la sequenza di quintuple che, data una parola x, costituiscono una computazione accettante di $NT(x)$, e quella parola è proprio y_x, anche se prima dobbiamo verificare la correttezza e quindi dobbiamo:

- verificare che y_x sia una sequenza di quintuple di NT che può eseguire su input x
- verificare che y_x corrisponda ad una computazione accettante
  se tutti questi casi sono verificati, allora posso concludere che $x \in L$

Per eseguire questa verifica costruiamo una macchina deterministica T, che chiameremo **verificatore**.
Quanto impiega $T(x, y_x)$ ad eseguire la verifica?

- poiché $L \in \mathbf{NP}$, se $x \in L$ allora esiste una computazione deterministica accettante di NT lunga $|x|^k$ passi - dove $\text{ntime}(NT, z) \leq |z|^k$ per ogni $z \in L$
- perciò $|y_x| \leq |x|^k$
- per verificare che y_x sia una sequenza di quintuple di NT, T impiega $O(|y_x|)$ passi
- per verificare che y_x corrisponda ad una computazione accettante di $NT(x)$, T deve simulare l'esecuzione delle quintuple descritte in y_x e dunque simula $|x|^k$ passi di NT e

impiega $O(|x|^k \cdot |y_x|) \subseteq O(|x|^{2k})$ passi
Quindi T impiega tempo polinomiale in $|x|$ per verificare che il genio abbia detto la verità.

Ricapitolando:
se $L \in \mathbf{NP}$ - e quindi è accettato da una macchina non deterministica NT tale che, per ogni $x \in L, \text{ntime}(NT, x) \leq |x|^k$ - se ho un genio in grado di suggerirmi, per ogni $x \in L$, la parola y_x che corrisponde ad una computazione accettante di $NT(x)$, allora posso costruire un **verificatore deterministico T** tale che, se gli do in input una parola x e la parola y_x che mi ha suggerito il genio, $T(x, y_x)$ accetta se e solo se $x \in L$ e il genio ha comunicato la parola y_x corretta e lo fa in tempo polinomiale in $|x|$.

Oss.
T è in grado di verificare che il genio non ha mentito solo se $x \in L$.
Se $x \notin L$ non c'è verso, infatti il genio non può trovare una parola che corrisponda ad una computazione accettante di $NT(x)$, ma per come abbiamo costruito T, se $x \notin L$, qualunque parola y ci venga indicata dal genio, $T(x, y)$ **rigetta!**

Siccome per ogni $x \in L, \text{ntime}(NT, x) \leq |x|^k$, allora posso fare in modo che, per ogni $x \in L$ e per ogni y tale che $|y| \leq |x|^k, \text{dtime}(T, x, y) \leq |x|^{hk}$.

Un po' di osservazioni

Nell'enunciato del teorema si parla dell'esistenza di una $y_x \in \{0,1\}^*$, ma la y_x che abbiamo tirato fuori nella dimostrazione mica è una parola in $\{0,1\}^*$, ma sappiamo bene come codificare una parola in binario, ed abbiamo giá parlato di come trasformare una macchina di Turing definita su un alfabeto generico in una macchina di Turing definita sull'alfabeto $\{0,1\}$ in modo tale che esse siano polinomialmente correlate.
Successivamente quel che ci chiediamo é: "$x \in L$"?
Se il Genio risponde di "si", noi non gli crediamo, allora, per dimostrarci che ha detto la veritá, ci comunica la parola y_x che poi noi successivamente verificheremo.

Per questo, se $x \in L$, y_x prende il nome di **Certificato** per x

Dimostrazione $\Longleftarrow$

Teorema 9.1: Un linguaggio $L \subseteq \Sigma^*$ appartiene ad NP se e solo se:

Esistono una macchina di Turing deterministica T e due costanti $h, k \in \mathbb{N} : \forall x \in \Sigma^*$,

$$x \in L \iff \exists y_x \in \{0,1\}^* : |y_x| \leq |x|^k \wedge T(x, y_x) \text{ accetta} \wedge dtime(T, x, y_x) \in O(|x|^h)$$

Dobbiamo dimostrare la seconda parte del teorema:

Dato L,

Se esistono una macchina di Turing deterministica T e due costanti $h, k \in \mathbb{N} : \forall x \in \Sigma^*$,

$x \in L \iff \exists y_x \in \{0,1\}^* : |y_x| \leq |x|^k \wedge T(x, y_x)$ accetta $\wedge\ dtime(T, x, y_x) \in O(|x|^h)$

Bisogna dimostrare che $L \in NP$, ovvero che esistono una macchina di Turing non deterministica NT e un intero a:

$\forall x \in L,\ NT(x)$ accetta e $ntime(NT, x) \in O(|x|^a)$

$\forall x \notin L,\ NT(x)$ non accetta

E come dimostriamo che esistono NT ed a?

1. Costruiamo NT (sfruttando la nostra conoscenza delle parole in L ed usando T)
2. Dimostriamo che NT accetta L
3. Dimostriamo che, sulle parole di L, NT opera in tempo polinomiale

Fase 1.

Cosa sappiamo sulle parole di L?

$x \in L \iff \exists y_x \in \{0,1\}^* : |y_x| \leq |x|^k \wedge T(x, y_x)$ accetta $\wedge\ dtime(T, x, y_x) \in O(|x|^h)$

T, h, k li conoscimo, allora costruiamo una macchina NT che opera in due fasi:$

Con input x:

1. NT sceglie non deterministicamente una parola binaria y di lunghezza $|y| \leq |x|^k$
2. NT invoca $T(x, y)$ e, se $T(x, y)$ accetta entro $O(|x|^h)$ passi, allora NT accetta

Oss.

$f(n) = n^k$ é una funzione time-costructible - sia T_f il trasduttore che la calcola, in unario, con $dtime(T_f, n) \in O(n^k)$

Vediamo ora nel dettaglio la Fase 1:

Con input x:

"FI/MOD II/img/img18.png" non trovato.

Assumiamo che se $x \in L, T$ accetta entro $c|x|^h \in O(|x|^h) passi$

- anche $g(n) = cn^h$ è una funzione time-costructible
- sia T_g il trasduttore che la calcola in unario, con $dtime(T_g, n) \in O(cn^h)$

Vediamo ora nel dettaglio la Fase 2:

Con input y:

“FI/MOD II/img/img19.png” non trovato.

Se $x \in L$ allora esiste $y_x \in \{0, 1\}^* : |y_x| \leq |x|^k \wedge T(x, y_x)$ accetta

- Allora *esiste una sequenza di scelte* nella Fase 1 che genera proprio y_x
- Allora, nella Fase 2, $T(x, y_x)$ accetta entro $c|x|^h$
- Allora, anche la computazione deterministica di $NT(x)$ corrispondente alla sequenza di scelte che ha generato y_x accetta

Questo dimostra che, se $x \in L$, allora $NT(x)$ accetta

Se $x \notin L$ allora non esiste alcuna $y_x \in \{0, 1\}^* : |y_x| \leq |x|^k \wedge T(x, y_x)$ accetta

- Allora, *qualunque sia la sequenza di scelte* nella Fase 1 per generare una parola y, nella Fase 2, $T(x, y_x)$ non accetta
- Questo significa che nessuna computazione deterministica di $NT(x)$ accetta

Questo dimostra che, se $x \in L$, allora $NT(x)$ non accetta

3. Dimostriamo che, sulle parole di L, NT opera in tempo polinomiale

Fase 1:

“FI/MOD II/img/img18.png” non trovato.

Calcolare B richiede $O(|x|^k)$ passi

Il ciclo *while* esegue $|x|^k$ iterazioni, in ciascuna della quali

- Sceglie un valore in un insieme di dimensione costante e quindi, impiega un numero costante di operazioni
- Incrementa di 1 una variabile, ovviamente assumendo che questo abbia costo costante (ma non è così)

Quindi, complessivamente, il ciclo **while** esegue $O(B) = O(|x|^k)$ operazioni

Fase 2:

“FI/MOD II/img/img19.png” non trovato.

Calcolare A richiede $O(|x|^h)$ passi

il ciclo *while* esegue $c|x|^h$ iterazioni, in ciascuna delle quali,

- Simula l' esecuzione di una istruzione della computazione $T(x, y)$ (costo costante)
- Confronta lo stato in cui è entrato T con q_a (costo costante)
- E, se non è q_a, incrementa di 1 una variabile (assumendo che abbia costo costante)

Quindi, complessivamente, il ciclo *while* esegue $O(|x|^h)$ operazioni

Lezione 20 - Teorema di Cook-Levin

Come facciamo a mostrare come trasformare un **qualsiasi** problema in NP a SAT se i problemi in NP sono così diversi tra di loro?
Semplice, sfruttando l'unica cosa che hanno in comune, tutti appartengono ad NP, ovvero, sono accettati da una macchina di Turing non deterministica in tempo polinomiale

Consideriamo un problema generico $\Gamma \in NP$ e sia $L_\Gamma \subseteq \{0,1\}^*$ il linguaggio che contiene la codifica ragionevole delle istanze **sì** di Γ e cerchiamo di descrivere sotto forma di espressione booleana il predicato "$x \in L_\Gamma$"
Questa dimostrazione che stiamo per affrontare è differente da quella che si trova sulle dispense

Dim.
Sia NT_Γ una macchina di Turing non deterministica ad un nastro che decide L_Γ in tempo polinomiale, ossia, esiste un polinomio p:
$\forall x \in \{0,1\}^*$

- $ntime(NT_\Gamma) \leq p(|x|)$
- $NT_\Gamma(x) = q_A$ se $x \in L_\Gamma$
- $NT_\Gamma(x) \neq q_A$ se $x \neq L_\Gamma$
  L'affermazione "$x \in L_\Gamma$" è logicamente equivalente all'affermazione:
  $\gamma(x) =$ "x è scritto sul nastro di NT_Γ
- *e* la testina di NT_Γ è posizionata sul primo carattere di x
- *e* NT_Γ è nel suo stato iniziale
- *e* esiste una sequenza di al più $p(|x|)$ quintuple di NT_Γ che possono essere eseguite una di seguito all'alòtra e portano la macchina nello stato q_A"
  cioè $x \in L_\Gamma \iff \gamma(x)$ è vera

Non resta che descrivere una **computazione di NT_Γ che ha iniziato con x scritto sul suo nastro** e dato che ogni computazione di una macchina di Turing è una sequenza di stati globali, per costruire $E(x)$ è necessario introdurre le variabili booleane che descrivono, per ogni passo di t della computazione lo stato globale in cui si troverebbe NT_Γ al passo t della computazione NT_Γ :

- Insieme N di variabili booleane che permettono di rappresentare il carattere contenuto in ciascuna cella del nastro di lavoro di NT_Γ
- Insieme M di variabili booleane che permettono di rappresentare lo stato interno di NT_Γ

- Insieme R di variabili booleane che permettono di rappresentare la cella del nastro di lavoro sulla quale è posizionata la testina NT_Γ
  Analizziamo queste variabili:
  Partiamo con l'insieme M.
  Sia $Q = \{q_0, q_1, q_2, \dots, q_k\}$ l'insieme degli stati di NT_Γ
  Con q_0 stato iniziale, $q_1 = q_A$ e $q_2 = q_R$
  Questo insieme M, insieme ad una porzione E_M dell'espressione $E(x)$ che stiamo costruendo, servono a descrivere in quale stato interno si trova NT_Γ ad ogni passo della computazione $NT_\Gamma(x)$: quello che vogliamo è che,
- **Ogni volta che i valori assegnati alle variabili in M fanno assumere ad E_M il valore vero**
- **Osservando i valori assegnati alle variabili contenute in M, dobbiamo essere in grado di rispondere a domande del tipo "è q_4 lo stato interno di NT_Γ al passo 25 della computazione $NT_\Gamma(x)$?"**
  Per ogni passo t $(0 \leq t \leq p(|x|))$, e per ogni $i \in \{0, 1, \dots, k\}$, M contiene una variabile booleana M_i^t :

  $$M = \{M_i^t : 0 \leq t \leq p(|x|) \wedge i \in \{0, 1, \dots, k\}\}$$

  Questo significa: **assegnando a M_i^t il valore vero rappresentiamo il fatto che, al passo t della computazione $NT_\Gamma(x)$, la macchina NT_Γ si trova nello stato q_i**

Lezione 21 - NP completezza

Teorema di Cook-Levin e la struttura di NP

Partiamo dalla domande precedente:
Fra i problemi in NP che non si riesce a collocare in P, ce ne sono alcuni più "difficili" rispetto ad altri?

Il *Teorema di Cook-Levin* ci dice che NP contiene un problema $NP-$completo (SAT), e dato che sappiamo che i problemi completi per una classe sono i problemi più "difficili" fra i problemi in quella classe, il *Teorema di Cook-Levin* ci dice che SAT è uno dei problemi più difficili in NP, perchè sappiamo che se SAT appartenesse a P, allora anche ogni altro problema in NP apparterrebbe a P, perchè ricordiamo, P è chiuso rispetto alla riducibilità polinomiale.
Ma il *Teorema di Cook-Levin* ci dice molto di più!

Il Teorema e la congettura

Sappiamo della congettura $P \neq NP$
Bene, arriva qualcuno e dimostra che $P = NP$ e lo fa **descrivendo un algoritmo deterministico che decide SAT in tempo polinomiale**
Ma a cosa ci serve sapere che $P = NP$?
Beh molto semplice, prendendo un problema in NP so che *esiste* un algoritmo deterministico che lo decide in tempo polinomiale.
Ma che ci faccio con l'*esistenza*?

Se sapessi che $P = NP$

A che mi serve sapere che, siccome un problema si trova in NP e $P = NP$, un algoritmo deterministico polinomiale che lo decide *esiste*, se io un tale algoritmo non riesco a progettarlo?
In realtà il *teorema di Cook-Levin* fa molto di più che dimostrare che SAT è $NP - completo$
La dimostrazione del teorema di *Cook-Levin* è la descrizione di un algoritmo deterministico che trasforma le istanze di un qualunque problema NP in istanze di SAT.
Se abbiamo un algoritmo deterministico polinomiale che decide SAT allora la dimostrazione del teorema di Cook-Levin ci mostra come costruire un algoritmo

polinomiale che decide qualunque problema in NP
Vediamo come avviene:

Supponiamo di avere un algoritmo deterministico polinomiale che decide SAT e lo chiamo T_{SAT}, allora
$\forall y \in \{0,1\}^*, dtime(T_{SAT}, y) \leq |y|^k$
Un problema decisionale Γ e dimostro che $\Gamma \in NP$, quindi progetto una macchina non deterministica NT_Γ che lo decide in tempo polinomiale.
Allora, considero il seguente algoritmo: con input $x \in \{0,1\}^*$

1. Costruisce $E(x)$ (COme in Cook-Levin)
2. Esegue $T_{SAT}(E(x))$: se termina in q_A allora accetta, altrimenti q_R
   Basandoci sulla dimostrazione di Cook-Levin, questo algoritmo decide L_Γ, inoltre, richiede tempo polinomiale $|x|$, infatti:
3. Tempo polinomiale $|x|$
4. Tempo $|E(x)|^k$ dove $E(x)$ ha lunghezza polinomiale in $|x|$
   Allora, abbiamo costruito un algoritmo deterministico polinomiale che decide Γ **Nell'ipotesi di avere un algoritmo deterministico polinomiale che decide SAT**

Il Teorema e la congettura

Quindi se si dimostrasse che $P = NP$ e si trovasse un algoritmo deterministico polinomiale che decide SAT, allora il Teorema di Cook-Levin ci permetterebbe di costruire un algoritmo deterministico polinomiale per decidere qualunque problema in NP

Ma, se invece si dimostrasse il contrario? Ovvero che $P \neq NP$
In questo caso sapremmo che $SAT \notin P$ ed ogni volta che riuscissimo a dimostrare che un problema è $NP - completo$, sapremmo che quel problema $\notin P$, quindi i problemi $NP - completi$ sono i problemi sepratori fra P e NP nell'ipotesi $P \neq NP$

Da problema a problema

Per fortuna abbiamo uno strumento che ci aiuta ogni volta nelle dimostrazioni, anche perchè se sono come quella di Cook-Levin sarebbe troppo lungo.

Dati 2 problemi Γ e Λ quando è che $\Gamma \preceq \Lambda$?
Facile, quando $L_\Gamma \preceq L_\Lambda$, dove L_Γ e L_Λ sono i linguaggi associati alle codifiche ragionevoli delle istanze *si* dei due problemi.
Allora $\Gamma \preceq \Lambda$ se $\exists f : \mathfrak{I}_\Gamma \to \mathfrak{I}_\Lambda$ tale che

- $f \in FP$

- x è un'istanza *si* di $\Gamma \iff f(x)$ è un'istanza *si* di Λ
  (Per semplicità, $x \in \Gamma$ "x è un'istanza si di Λ")

Teorema 9.3: Sia Γ un problema in NP. Se esiste un problema $NP-completo$ riducibile a Γ allora Γ è $NP-completo$

Sia Λ un problema $NP-completo$ tale che $\Lambda \preceq \Gamma$.
Poichè $\Lambda \preceq \Gamma$, esiste $f: \Im_\Lambda \to \Im_\Gamma$ tale che $f \in FP$ e $\forall y \in \Im_\Lambda, y \in \Lambda \iff f(y) \in \Gamma$
Poichè $\Lambda è NP-completo$, $\forall$ problema $\Delta \in NP$ si ha che $\Delta \preceq \Lambda$:

- $\exists g: \Im_\Delta \to \Im_\Lambda$ tale che $g \in FP$ e,
- $\forall x \in \Im_\Lambda, x \in \Delta \iff g(x) \in \Lambda$
  La composizione delle due funzioni g, f è una riduzione polinomiale da Δ a Γ
- Sia $x \in \Im_\Delta$: allora $x \in \Delta \iff g(x) \in \Lambda$ e inoltre $g(x) \in \Lambda \iff f(g(x)) \in \Gamma$
  Allora, chiamando h la composizione delle due funzioni, dimostro che h è una riduzione da Δ a Γ.
  Ma, quanto costa calcolare h?
  $g \in FP$, allora esistono un trasduttore T_g e una costante $k \in \mathbb{N}$ tali che $\forall x \in \Im_\Delta, T_g(x)$ calcola $g(x)$ e $dtime(T_g, x) \leq |x|^k$
  Dato che $T_g(x)$ deve anche scrivere il risultato $g(x)$ sul nastro di output, allora $|g(x)| \leq |x|^k$
  $f \in FP$, allora esistono un trasduttore T_f e una costante $c \in \mathbb{N}$ tali che, $\forall y \in \Im_\Lambda, T_f(y)$ calcola $f(y)$ e $dtime(T_f, y) \leq |y|^c$

Definiamo il trasduttore T_h a tre nastri, che calcola h: con $x \in \Im_\Delta$ scritto sul primo nastro T_h

1. Esegue la computazione $T_g(x)$ scrivendo il suo output $y = g(x)$ sul secondo nastro
2. Esegue la computazione $T_f(y)$ scrivendo il suo output $f(y)$ sul nastro di output
   $\forall x \in \Im_\Delta \; dtime(T_h, x) \leq |x|^k + |g(x)|^c \leq |x|^k + |x|^{kc} \leq 2|x|^{kc}$
   e questo dimostra che $h \in FP$
   Quindi, abbiamo dimostrato che $\Delta \preceq \Gamma$ poichè Δ è un qualunque problema in NP, questo ci prova che ogni problema in NP è riducibile polinomialmente a Γ
   Dall'appartenenza di Γ a NP segue che Γ è $Np-completo$ <SLIDE 10>