Primo semestre

1) Definizione di insieme numerabile. Quando un sott'insieme è numerabile?
Un insieme **A** si dice **numerabile** se esiste una funzione suriettiva f dall'insieme dei numeri naturali ad A. Ogni sott'insieme di un insieme numerabile è anch'esso numerabile.

Intuitivamente, un insieme è numerabile se esiste un cammino che passa per tutti gli elementi di quell'insieme.

La funzione f si dice anche funzione di numerazione.

---

2) Cosa dice il teorema di Cantor? Dimostrarlo.

Dato un insieme arbitrario A, NON esiste una *funzione suriettiva* che va da A all'insieme delle parti di A. Dunque, l'insieme delle parti **NON è numerabile**.

Dimostrazione:
Supponiamo per assurdo che questa funzione suriettiva f esista e sia definita come f: A $\rightarrow$ P(A). In questo modo stiamo dicendo che con A possiamo numerare P(A). Consideriamo poi il seguente insieme:

$$\Delta \subseteq A, \Delta := \{a \in A \mid a \notin f(a)\}$$

Ovvero l'insieme composto dagli elementi di A che non appartengono all'immagine di sé stessi.
Visto che Δ è un sott'insieme di A (quindi è un elemento dell'insieme delle parti) e f è suriettiva, deve esistere un elemento δ: $f(\delta) = \Delta$. Tuttavia, questo è assurdo in quanto:

$$\delta \in \Delta <=> \delta \notin f(\delta) <=> \delta \notin \Delta$$

---

3) Cosa dice il paradosso di Russel? [Non importante]
Sia $U = \{x \mid x \notin x\}$, ovvero l'insieme degli elementi che non sono contenuti in sé stessi.
Allora:

$$U \in U <=> U \notin U$$

---

4) Cosa si intende con problema della definibilità? [Non importante]
Il problemad della definibilità dice che la nozione di definibilità e dunque sempre relativa (ovvero dipendente dal linguaggio che si usa) e incompleta.

---

5) Quando una funzione è ricorsiva? Cosa si intende con funzioni primitive ricorsive?
Una funzione f è ricorsiva se il valore di f dipende dal valore di f su altri argomenti solitamente più semplici.

Le funzioni primitive ricorsive sono un sott'insieme delle funzioni ricorsive, che permettono di calcolare le funzioni totali (ovvero tutte le funzioni espresse da programmi che non hanno cicli illimitati).

---

6) Quando una funzione è primitiva ricorsiva?

Una funzione è primitiva ricorsiva se e solo se esiste una sequenza finita di funzioni f1,f2..fn tali per cui f=fn, ed ogni fi è una *funzione base* o si ottiene della composizione di esse o si ottiene dalla ricorsione primitiva partendo da quelle fj tali che j > i.

7) Quando un predicato è primitivo ricorsivo?
Si dice che P è un predicato primitivo ricorsivo se e solo se la sua funzione caratteristica c_P è primitiva ricorsiva

---

8) Cosa si intende con μ? Cos'è la minimizzazione limitata?
Con μ si intende la minimizzazione, ovvero il minimo intero progressivo che soddisfi un predicato P.

---

9) Cosa si intende con for-calcolabile? Le funzioni primitive ricorsive sono for-calcolabili?
Le funzioni primitive ricorsive sono tutte quelle for-calcolabili, ovvero esprimibili in un linguaggio imperativo del prim'ordine usando esclusivamente i costrutti di controllo di flusso, if-then-else il for e la chiamata ad una funzione ricorsiva.

---

10) Nel formalismo primitivo ricorsivo ho delle funzioni di ordine superiore?
No.

---

11) Perché non possiamo esprimere la funzione di Ackermann nel formalismo primitivo ricorsivo? **Cosa potremmo fare invece per fare in modo che la funzione di Ackermann sia esprimibile nel formalismo primitivo ricorsivo?**

La funzione di Ackermann `ack(z,x,y)` è una funzione a 3 parametri, ed è un esempio di funzione terminante che sappiamo non poter esprimere con i soli costrutti for e if-then-else, a causa della forma di ricorsione che usa.
In particolare, non può essere espressa da questo siccome è possibile dimostrare che ogni funzione f primitiva ricorsiva è limitata in tempo da un'istanza **parziale** (ovvero con uno z fissato) della funzione di Ackermann. Il fatto che sia parziale ne impedisce di essere espressa come formalismo primitivo ricorsivo.
Questo implica inoltre che possiamo scrivere un'istanza sola della funzione di ackermann, ma non possiamo inglobarle tutte (con il form. primitivo ricorsivo).

Per calcolare la funzione di Ackermann, possiamo estendere l'espressività del formalismo primitivo ricorsivo in due modi:

- Indebolendo il principio lessicografico di ricorsione strutturale, in modo che si accetti la ricorsione solo **quando l'ordinamento lessicografico diminuisce**. In questo modo però, non riusciamo a stabilire un limite superiore della catena delle chiamate.

- Manteniamo il sistema strutturale, ma estendiamo il sistema di tipi in modo da includere le funzioni di ordini superiore. Infatti, possiamo esprimere la funzione di Ackermann usando la funzione next, che itera un iteratore.
  Otteniamo così un nuovo linguaggio detto Sistema T.

---

12) Cos'è un interprete? Illustrare il problema dell'interprete nei formalismi totali.
Sia ϕ_n la funzione calcola dal programma P_n.
Un interprete per il nostro formalismo *L* è una funzione *I* che prende in input l'indice n di un programma e un input m, calcola il comportamento di P_n su m, ovvero:

$$I(n, m) = \phi_n(m)$$

Adesso, supponiamo per assurdo che esista un indice u tale per cui:

$$I(n, m) = \phi_u(n, m) = \phi_n(m)$$

Consideriamo la funzione:

$$f(x) = \phi_u (x, x) + 1 = \phi_x(x) +1$$

Da questo, ne deriva che siccome f(x) è esprimile nel formalismo *L*, dunque deve esistere un *i* tale che $\phi_i = f$. In questo modo, però, arriviamo ad un assurdo:

$$\phi_i(i) = f(i) = \phi_i(i) + 1$$

Da questo concludiamo che:
"Nessun formalismo totale è in grado di esprimere il proprio interprete".

Se il nostro formalismo non fosse totale, allora il programma P_i potrebbe divergere in entrambi i lati dell'equazione, e quindi entrambi i lati dell'equazione potrebbero essere indeterminati e non ci sarebbe l'assurdo.

---

13) Cosa dice la tesi di Church?

"Le funzioni calcolabili sono esattamente quelle intuitivamente calcolabili mediante una procedura effettiva di calcolo"
Seconda versione: La classe delle funzioni calcolabili coincide con la classe delle funzioni calcolabili mediante una macchina di Turing.

---

14) **Cos'è** e com'è fatta una macchina di Turing universale?

Una macchina di Turing universale è una speciale macchina di Turing che accetta in input una descrizione di una macchina di Turing M (intesa come programma), e prova a simularla su un determinato input m.

Ecco com'è fatta:

- La descrizione della MdT è rappresentata dalle quintuple del grafo associato alla funzione di transizione.
- Un unico nastro viene suddiviso in due parti da un carattere speciale, in modo che:
  - una parte contenga l'insieme delle quintuple di M
  - una parte contentente il "nastro di lavoro", dove inizialmente viene copiato l'intero input m.
- Lo stato interno è rappresentato da 3 registri Q, A, S, che sono usati per memorizzare rispettivamente:
  - Lo stato corrente della macchina M, inizializzato come $Q = q_0^M$
  - Il carattere di lettura/scrittura
  - Lo shift left/right della testina

Dopo che si legge un carattere *a* sul nastro di input, si rimpiazza *a* con un carattere speciale #, per ricordare la posizione della testina, e si memorizza a in A. A questo punto, si cerca nella parte del nastro che contiene le quintuple una quintupla che faccia con i vallori correnti di Q e A.

A questo punto, si modificano i registri in questo modo:

- Q = q'
- A = a'
- S = L/R

Se q' è finale, allora la computazione si arresta, altrimenti si torna a cercare il carattere # e lo si rimpiazza con il contenuto di A, si sposta la testina in base al valore di S e si ricomincai il ciclo.

---

15) Cosa dice la proprietà utm?

Esiste un indice *u* tale che per ogni x e per ogni y:

$$\varphi_u(x, y) = \varphi_x(y)$$

Questa proprietà indica che esiste un programma universale *u* tale per cui se io gli do un programma x con input y, allora calcolerà/interpreterà esattamente quel programma. Essenzialmente quindi, questa proprietà ci dice solo che tra le funzioni calcolabili ci sia anche la macchina universale, ovvero un interprete.

---

16) Cosa dice la proprietà smn?

Per ogni funzione parziale calcolabile f^{m+n} (m+n sono il numero di argomenti della funzione) esiste una funzione totale calcolabile s_n^m che, per ogni $\vec{x}_m$, $\vec{y}_n$ (ovvero i vettori di input x e y):

$$\varphi_{s_n^m(\vec{x}_m)}(\vec{y}_n) = f(\vec{x}_m, \vec{y}_n)$$

Essenzialmente ci dice che se una funzione *f* è calcolabile, allora io mi aspetto che tutte le istanze della *f* siano calcolabili (per questo s_n^m è totale). Ovvero, esiste una funzione s_n^m totale tale per cui, presa una porzione dell'input di x di *f*, calcola l'indice della funzione che calcola quella particolare istanza di *f*.
Quindi, le funzioni calcolabili sono chiuse rispetto la **valutazione parziale**.

---

17) Quando una enumerazione è detta accettabile?
Una enumerazione è detta accettabile quando gode sia della proprietà smn che utm.

---

18) Cos'è la currificazione?
La currificazione è una tecnica che ci permette di decomporre una funzione di arietà *n* in una sequenza di *n* funzioni da un solo argomento.

---

19) Quando due funzioni di enumerazioni sono reducibili? Quando sono equivalenti? **Qual è la relazione di questa proprietà con le enumerazioni accettabili?**
Siano date due funzioni di enumerazione $\varphi, \psi : N \to A$

1. ψ è **riducibile** a φ (in simboli: $\psi \leq \varphi$) se esiste una funzione totale calcolabile *f* tale che, per ogni numero naturale *n*, $\psi_n = \varphi_{f(n)}$.
2. ψ è **equivalente** a φ (in simboli: $\psi \equiv \varphi$) se $\psi \leq \varphi$ e $\varphi \leq \psi$.

Tutte le **enumerazione accettabili** di funzioni parziali ricorsive sono equivalenti fra loro.

---

20) Cos'è il predicato T di Kleene?

Siccome le funzioni parziali sono potenzialmente divergenti, anche l'interprete di queste è potenzialmente divergente. Tuttavia, è possibile ottenere una approssimazione terminante dell'*interprete* delle funzioni parziali (detta **interprete bound**) che essenzialmente limita le risorse di calcolo in modo da rendere la funzione divergente terminante.

L'idea è quella di chiedere all'interprete di simulare la computazione della funzione ***i*** sull'input ***n*** con risorse limitate, ovvero in un tempo massimo t e osservare lo stato della computazione a seguito dell'esaurimento delle risorse.

Questa idea si può formalizzare con il predicato T di Kleene, che dice che:

Esiste un predicato $T(i, n, k, t)$ tale che

1. $\varphi_i(n) = k \Leftrightarrow \exists t \geq k, T(i, n, k, t) = \text{true}$
2. la funzione caratteristica di T è calcolabile

Ovvero, il predicato T ci restituisce *vero* se il programma di indice i con input n termina in tempo t e fornisce risultato k. Ovvero, ***$\varphi_i(n) = k$*** solo se ci sono risorse sufficienti affinché il predicato restituisca true.

---

21) Spiegare la forma normale del predicato di Kleene. Cosa deduciamo da questo?
La forma normale di Kleene:
Per ogni i, n

$$\varphi_i(n) = first(\mu_{<k,t>}\ T(i, n, k, t))$$

Che ci da una spiegazione più fine di un interprete che tiene anche conto del costo computazionale. Infatti la funzione T può essere calcolata in tempo O(t), visto che abbiamo dato un upper bound delle risorse utilizzabili, quindi il predicato T è **primitivo ricorsivo**.
First è una proiezione che restituisce il primo elemento della coppia.

Da questo predicato, deduciamo che ogni funzione calcolabile è esprimibile come una combinazione di for e while.

---

22) Illustrare l'halting problem. È un esempio di quale tipo di funzione?
Sia *h(i, x)* la funzione del test di terminazione. Questo è definito come:

$$h(i,x) = \begin{cases} 1 & \text{se } \varphi_i(x)\downarrow \\ 0 & \text{se } \varphi_i(x)\uparrow \end{cases}$$

Restiuisce 1 se il programma converge, e 0 se il programma diverge.

Questa funzione è definibile (l'abbiamo appena definita…) e supponiamo che sia calcolabile.
Possiamo allora costruire un'altra funzione *f* definita come

$$f(x) = \begin{cases} 1 & se\ h(x,x) = 0 \Leftrightarrow \varphi_x(x)\uparrow \\ \uparrow & se\ h(x,x) = 1 \Leftrightarrow \varphi_x(x)\downarrow \end{cases}$$

Questa funzione, se *h* è calcolabile, è anch'essa perfettamente calcolabile. Infatti, prende un solo argomento, e restituisce 1 se la funzione del programma i diverge su input i, altrimenti se esso converge la *f* non è definita.

Se *f* è calcolabile, allora esisterà un $\varphi_m = f$. Ma allora avremo che:

$$\varphi_m(m) = \begin{cases} 1 & se\ h(m,m) = 0 \Leftrightarrow \varphi_m(m)\uparrow \\ \uparrow & se\ h(m,m) = 1 \Leftrightarrow \varphi_m(m)\downarrow \end{cases}$$

Ovvero che la funzione φ_m converge se e solo se φ_m diverge. CHE È UN ASSURDO!

È il classico esempio di una funzione definibile, ma non calcolabile.

---

23) Illustrare il problema della totalità.
Il problema della totalità consiste nel stabilire se possiamo calcolare una funzione che ci dice quando una funzione è totale oppure no.
Sia la funzione *total* :

$$total(i) = \begin{cases} 1 & se\ \varphi_i\ è\ totale \\ 0 & altrimenti \end{cases}$$

Sia poi una funzione calcolabile k(x, y) tale per cui:

$$k(x, y) = \begin{cases} 0 & se\ \varphi_x(x)\downarrow \\ \uparrow & altrimenti \end{cases}$$

Per continuare, usiamo una tecnica chiamara ***riduzione***.
Essenzialmente, data una funzione g diciamo che ***g ≤ f*** se esiste una funzione h tale per cui:

$$\forall x\ g(x) = f(h(x))$$

Se sappiamo che g non è calcolabile, mentre h lo è, allora f non sarà calcolabile.

Per smn esiste un h totale calcolabile tale che

$$\varphi_{h(x)}(y) = k(x, y)$$

Dunque, sappiamo che total prende in input l'indice della funzione che deve stabilire se è totale o meno, perciò applichiamo *total* alla funzione con indice *h(x)*.
Otteniamo così:

$$total(h(x)) = \begin{cases} 1 & se\ \varphi_x(x)\downarrow \\ 0 & se\ \varphi_x(x)\uparrow \end{cases}$$

È la stessa funzione dell'halting problem.

Sappiamo che h è sicuramente calcolabile (per smn). Siccome dobbiamo ridurre ad una funzione non calcolabile (che in questo caso è g), l'unica funzione non calcolabile DEVE essere *total*.
Dunque, total non è calcolabile.

---

24) Illustrare il problema dell'equivalenza fra programmi.

Il problema dell'equivalenza fra programmi consiste nello stabilire se esistono due programmi che calcolino la stessa funzione. Ovvero, se esiste una funzione eq tale per cui:

$$eq(i, j) = \begin{cases} 1 & se\ \varphi_i \approx \varphi_j \\ 0 & altrimenti \end{cases}$$

e che sia calcolabile.

Siano poi due funzioni. La prima è φ_m, ed è una funzione costante tale che è = 0 per ogni input.
Sia poi la funzione

$$\varphi_{h(x)}(y) = \begin{cases} 0 & se\ \varphi_x(x)\downarrow \\ \uparrow & se\ \varphi_x(x)\uparrow \end{cases}$$

Sappiamo che questa è calcolabile, e quindi h(x) è totale calcolabile.
Tuttavia, otteniamo così che:

$$eq(h(x), m) = \begin{cases} 1 & se\ \phi_{h(x)}(y) = 0 \iff \phi_x(x)\downarrow \\ 0 & altrimenti \iff \phi_x(x)\uparrow \end{cases}$$

Che è la funzione dell'halting problem, e quindi sappiamo che non è calcolabile.

---

25) Quando un insieme è ricorsivo? Alcuni esempi? Alcune proprietà degli insiemi ricorsivi?

Un insieme si dice **ricorsivo** quando la funzione caratteristica è calcolabile.
Esempi:

1. Insiemi definiti da predicati primitivi ricorsivi
2. Insieme vuoto e l'insieme N, dove la loro f caratteristica è 0 e 1.
3. Ogni insieme finito (siccome ogni insieme finito è calcolabile siccome è definito attraverso una funzione definita per casi)
4. Insieme dei numeri primi, e l'insieme dei numeri pari

Gli insiemi ricorsivi sono chiusi per negazione, intersezione e unione.

---

26) Cos'è un insieme ricorsivamente enumerabile?

Un insieme si dice ricorsivamente enumerabile se è vuoto, oppure se è codiminio di una funzione totale calcolabile, detta **funzione di enumerazione**.

Ogni insieme ricorsivo è anche ricorsivamente enumerabile (MA NON VICEVERSA), e quindi possiamo associargli una funzione di enumerazione.

---

27) Spiegare il legame fra enumerazioni crescenti ed insiemi ricorsivi.

Un insieme A è ricorsivo se e solo se può essere enumerato in modo crescente.

Questo perché essenzialmente, se facessimo una enumerazione non crescente, la nostra funzione di enumerazione potrebbe essere non totale (quindi parziale).

---

28) Dimostrare che un insieme A è ricorsivo se e solo se sia A che !A sono r.e. (Completezza)

→) Dobbiamo dimostrare che se A è ricorsivo allora A e !A sono r.e.
È banale, siccome sappiamo che gli insiemi ricorsivi sono chiusi per la negazione.

←) Dobbiamo dimostrare che se A e !A sono r.e., allora A è ricorsivo.
Siano f,g le funzioni di *enumerazione* di A e !A rispettivamente. Sia poi una funzione ausialiaria h tale per cui :

$$\begin{cases} h(2x) & = f(x) \\ h(2x+1) & = g(x) \end{cases}$$

Ovvero, per i numeri pari si comporta come f e per quelli dispari si comporta come g.
Dunque, possiamo facilmente calcolare la funzione caratteristica di A in questo modo:

$$c_A(n) = pari(\mu_y(h(y) = n))$$

Ovvero si cerca il più piccolo y tale per cui h(y) = n, ovvero che enumera A. Se è pari, allora $n \in A$, altrimenti se è dispari $n \notin A$.

---

29) Come si differenziano gli insiemi r.e. e ricorsivi riguardo la loro enumerabilità?

Negli insiemi ricorsivi, determinare se un elemento appartiene ad un insieme ricorsivo partendo dalla sua funzione di enumerazione è semplice (siccome sono enumerabili in ordine crescente), mentre negli insiemi ricorsivamente enumerabili costruire la funzione caratteristica partendo dalla funzione di enumerazione non è sempre possibile.

---

30) Quando un insieme è semidecibile?

Un insieme A è semidecidibile quando siamo in grado di dire in tempo finito se x ∈ A, ma non siamo in grando di stabilire in tempo finito se x ∉ A.

Nel nostro caso, gli insiemi r.e. sono quelli semidecibili, mentre quelli decidibili sono quelli ricorsivi.

---

31) Enunciare e dimostrare le caratterizzazioni equivalenti degli insiemi r.e.

Sia A un insieme di numeri naturali. Le seguenti affermazioni sono equivalenti:

1. A=∅ ∨ ∃f : A=*cod*(*f*), dove f totale calcolabile. (Questa corrisponde alla definizione di insieme r.e.)
2. ∃g : A=*dom*(*g*), dove g parziale calcolabile. (Ovvero un inisieme è r.e. sse esiste una funzione che ha tale insieme come dominio di convergenza.)
3. ∃h : A=*cod*(*h*), dove h parziale calcolabile (Equivalente al punto 1, ma si elimina la condizione in cui A può essere nullo, siccome adesso h è parzialmente calcolabile, ovvero se scegliamo come h la funzione che diverge sempre allora abbiamo che A = ∅).

Dimostrazione:
1 → 2) Se A = ∅, allora basta prendere g come quella funzione che diverge sempre. Altrimenti, sia A = cod(f), allora calcoliamo:

$$g(x) = \mu_y(f(y) = x)$$

quindi *g* converge solo su quegli x che sono enumerati da *f*, e diverge altrimenti.

2 → 3) Sia A = dom(g), allora consideriamo:

$$h(x) = x + 0 \cdot g(x)$$

In questo modo, se x ∈ dom(g), allora ce lo scordiamo andiamo ad eliminarlo (con lo zero), e ritornando x ad h. Altrimenti, diverge.

3 → 1) Se A = ∅, siccome possiamo prendere come h la funzione che diverge sempre. Altrimenti supponiamo che h = φ_i e che *a* ∈ A. Consideriamo così *f* definita come:

$$f(\langle x,k,s\rangle) = \begin{cases} k & se\ T(i,x,k,s) = 1 \\ a & altrimenti \end{cases}$$

L'idea e' di utilizzare il predicato di Kleene per limitare le risorse. Allora se con risorse *s*, h(x) = k, allora k ∈ A e quindi lo possiamo enumerare, altrimenti ritorniamo *a* che appartiene comunque ad A.
In questo modo, *f* ritorna solo elementi in A, e li copre tutti, in quanto possiamo incrementare *s*, rendola così totale calcolabile.

---

32) Fare un esempio di insieme r.e. che non è ricorsivo. Perché?

$$K = \{x \mid \varphi_x(x)\downarrow\ \}.$$

Questo insieme è r.e. siccome la funzione k = $\varphi_x(x)$ è *parziale* calcolabile, e K = dom(k). Tuttavia, la funzione caratteristica è l'halting problem, quindi non è calcolabile.

Inoltre, **!K non è r.e.**, altrimenti K sarebbe ricorsivo e sappiamo che non lo è.

---

33) In cosa consiste il teorema della proiezione? Dimostrarlo.

Un insieme A è r.e. se e solo se esiste un insieme B ricorsivo tale che:

$$A = \{m \mid \exists n, \langle n, m\rangle \in B\}$$

Ovvero se A è la proiezione di un insieme ricorsivo.

Dimostrazione:

←) Vogliamo dimostrare che se esiste un insieme B ricorsivo per cui A è la sua proiezione, allora A è r.e. .

Sia c_B la funzione caratteristica di B. Vogliamo trovare una funzione di semidecisione *f* tale per cui il suo dominio sia A. Questa è definita come:

$$f(m) = \mu_n, c_B(\langle n, m\rangle) = 1$$

Sappiamo che C_B è totale calcolabile.

ovvero cerco il più piccolo n tale che la coppia <n, m> sta in B. Se la trovo, allora m ∈ A, altrimenti cerco all'infinito.

→) Vogliamo dimostrare che se A è r.e., allora esiste un insieme B ricorsivo per cui A è la sua proiezione.

Sappiamo che A è r.e., quindi assumiamo che A = *dom*(φ_i), ovvero sia il dominio di convergenza della funzione φ_i . Dunque, $m \in A \iff \varphi_i(m)\downarrow$ (converge), ovvero se e solo se esiste un n tale per cui T(i, m, n) = 1 [T è il predicato ternario di Kleene], dove n è il tempo (stiamo cercando il tempo sufficiente). Basta poi porre:

$$B = \{\langle n, m\rangle \mid T(i, m, n) = 1\}$$

---

34) Dimostrare che gli insiemi r.e. sono chiusi per intersezione e unione.

→ Unione:

Sia A = cod(f), B = cod(g), e f,g sono parziali calcolabili. Allora definiamo una funzione h in modo che A ∨ B = cod(h), dove

$$\begin{cases} h(2x) & = f(x) \\ h(2x+1) & = g(x) \end{cases}$$

La nostra funzione è calcolabile, e il suo codominio è l'unione fra A e B. Dunque, l'insieme ottenuto è r.e.

→ Intersezione: Prendiamo h(x) = f(x) * g(x), dove A = dom(f), e A ∧ B = dom(h). Infatti h converge solo quando g e f convergono.

---

35) Qual è la proprietà di chiusura rispetto alle funzione degli insiemi ricorsivi?

La controimmagine di un insieme ricorsivo via una funzione totale calcolabile è un insieme ricorsivo.

Ovvero, sia f totale calcolabile, A ricorsivo: $f^{-1}(A)$ è ricorsivo.

Dimostrazione [ignora]

Sia c_A la funzione caratteristica di A. Sappiamo che $x \in f^{-1}(A) \iff f(x) \in A$, quindi $f^{-1}(A)$ non è altro che $c_A \circ f$, quindi per vedere se un elmento è nella contro immagine basta fare $c_A(f(x))$

---

36) Quando una proprietà si dice estensionale?

Una proprietà si chiama estensionale se è soddisfatta per tutti i programmi che la calcolano la stessa funzione. Ovvero, è una proprietà in relazione alla funzione calcolata (estensione) e non al modo in cui essa viene calcolata (intensione).

Formalmente:

$$i \in A \ \wedge \ \varphi i = \varphi j \rightarrow j \in A.$$

---

37) Enunciare e dimostrare il teorema di Rice

Una proprietà estensionale è decibile solo se banale, ovvero se sempre vera o sempre falsa.

Dimostrazione:

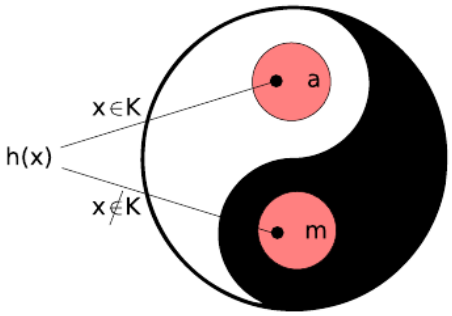
- Sia *c* la funzione caratteristica del predicato (ovvero della proprietà) estensionale A, e supponiamo che la proprietà non sia banale.
- Sia *m* un indice per una funzione ovunque divergente.

Siccome *c* non è banale, allora esiste sicuramente un indice di un programma *a* tale che $c_A(m) \neq c_A(a)$, ovvero esiste sicuramente un programma per cui la proprietà non vale.

Cerchiamo allora un h calcolabile tale che sia:

$$\phi_{h(x)} = \begin{cases} \phi_a & se\ x \in K \iff \phi_x(x)\downarrow \\ \phi_m & se\ x \notin K \end{cases}$$

Avremo questa situazione:



Se il programma x converge, allora h(x) è nella regione bianca (assieme a tutti i programmi che calcolano la stessa funzione di *a*), altrimenti è nella regione nera (assieme a tutti i programmi che divergono come m).

Per smn, possiamo costruire una funzione calcolabile che faccia esattamente ciò che dice la funzione $\varphi_{h(x)}$, ovvero:

$$\phi_{h(x)}(y) = \phi_x(x);\ \phi_a(y)$$

Questa funzione si comporta come *a* se il programma x con input x converge, altrimenti diverge per ogni input y.
A sto punto calcoliamo *c* su h (ovvero se vale la prorietà c su h), e otterremo:

$$c(h(x)) = \begin{cases} c(a) & se\ \phi_x(x)\downarrow \\ c(m) & altrimenti \end{cases}$$

Che ci dice esettamente se il programma termina o no (1 se termina, 0 se non termina o viceversa), e avremo risolto il problema della terminazione, che è un assurdo. Il fatto che sia non banale genera questo assurdo.

---

38) Quando un insieme è monotono?

Un insieme estensionale si dice monotono se per ogni i,j

$$i \in A \wedge \varphi_i \subseteq \varphi_j \ \rightarrow \ j \in A$$

Ovvero, una proprietà P è monotona quando avendo una funzione f che soddisfa tale proprietà, e una sua estensione g, anche g soddisferà quella proprietà.

---

39) Cos'è una restrizione? Quando un insieme è compatto?
Una restrizione di una funzione f sull'insieme A [notazione f/A(n)] è definita come una funzione che si comporta come f solo su gli elementi di A, altrimenti diverge.

Se una proprietà P è compatta per f, allora esiste almeno un insieme finito A tale per cui P è vera anche per la *restrizione* di *f* su A.

Formalmente [non importante]:
Un insieme estensionale A si dice compatto se vale che:

$$i \in A \rightarrow \exists j : \varphi_j \text{ ha grafo finito} \wedge \varphi_j \subseteq \varphi_i \wedge j \in A$$

Spiegazione grafo di una funzione a pag.24

---

40) Teorema di Rice-Shapiro, monotonia. [hint: parallelismo]

Il teorema di Rice-Shapiro, riguardo la monotonia, spiega che ogni insieme estensionale A r.e. è monotono.

Dimostrazione
Per dimostrarlo, dobbiamo mostrare che se un insieme estensionale A non è monotono, allora non può essere r.e. .
Siano quindi due indici i,j tali che $i \in A$, ma $j \notin A$ nonostante $\varphi_i \subseteq \varphi_j$ (negazione monotonia).
Consideriamo la funzione:

$$\phi_{f(x)}(y) = \phi_i(y) \mid (\phi_x(x); \phi_j(y))$$

Essenzialmente, se $\varphi_x(x)$ diverge, si comporta come φ_i , altrimenti si comporta come φ_j .
La definizione di questa funzione che abbiamo costruito sarà:

$$\phi_{f(x)} = \begin{cases} \phi_j & se\ x \in K \\ \phi_i & se\ x \notin K \end{cases}$$

Da questo, e dal fatto che A è estensionale, concludiamo che:

$$f(x) \in A \Leftrightarrow x \in \overline{K}$$

Siccome, se x sta in K allora si comporta come j, che non sta in A, quindi $f(x) \notin A$, altrimenti se invece $x \in !K$ allora si comporta come A.

Se è A però fosse r.e., allora gli permetterebbe di semidecidere l'appartenenza a !K, che sappiamo non essere r.e., e quindi è assurdo.

---

41) Teorema di Rice-Shapiro, compattezza. [hint: passi]

Il teorema di Rice-Shapiro, riguardo la compattezza, spiega che ogni insieme estensionale A r.e. è compatto.

Dimostrazione
Per dimostrarlo, dobbiamo mostrare che se A non è compatto, allora non può essere r.e.,
Quindi, abbiamo un insieme A e un indice i tale che $i \in A$, tuttavia per ogni j tale che $\varphi_j \subseteq \varphi_i$ si ha $j \notin A$.
Consideriamo la funzione:

$$\phi_{f(x)}(y) = \begin{cases} \uparrow & se\ \phi_x(x) \downarrow\ in\ meno\ di\ y\ passi \\ \phi_i(y) & altrimenti \end{cases}$$

Abbiamo due casi da considerare:

- Se $x \notin K$ (e quidni $\varphi_x(x)$ diverge), allora $\varphi_x(x)$ diverge sempre, quindi $\varphi_{f(x)}(x) = \varphi_i(x)$, quindi $f(x) \in A$.
- Altrimenti, prima o poi arriveremo ad valore di y pari a *t* tale per cui $\varphi_x(x)$ converge sempre.
  Quindi fino a *t* si ha che $\varphi_{f(x)}$ si comporterà come φ_i , dopodiché divergerà sempre (siccome $\varphi_{(x)}(x)$ convergerà per meno di y passi). Quindi f(x) calcola una restrizione finita di i, e per ipotesi allora $f(x) \notin A$.

Concludiamo nuovamente che:

$$f(x) \in A \Leftrightarrow x \in \overline{K}$$

Come abbiamo spiegato nella domanda prima, è assurdo.

---

42) Teorema del punto fisso di Kleene.

Il teorema dice che per ogni funzione totale calcolabile ***f*** esiste un ***m*** tale per cui:

$$\phi_{f(m)}(n) = \phi_m(n) \quad \forall n$$

Dimostrazione:
Consideriamo una funzione g definita come

(1) $$g(x,y) = \phi_{f(\underbrace{\phi_x(x)}_{autoapplicazione})}(y)$$

Applichiamo la tecnica dell'autoapplicazione. In questo modo facciamo sì che il programma richiami sé stesso.

La funzione g è parziale calcolabile, quindi per smn esiste un h totale calcolabile tale che

(2) $$\phi_{h(x)}(y) = g(x,y)$$

Essendo *h* una funzione totale calcolabile, esisterà un indice p per essa. Poniamo

(3) $$h(p) = \phi_p(p) = m$$

m è sicuramente definito in quanto h è totale. Allora per ogni y abbiamo che:

$$\phi_{f(m)}(y) = \phi_{f(\phi_p(p))}(y) = g(p,y) = \phi_{h(p)}(y) = \phi_m(y)$$

Per (3) — Per (1) — Per (2) — Per (3)

---

43) Secondo teorema di ricorsione di Kleene [NON IMPORTANTE, SALTA]

Mentre il punto fisso normale considera una funzione unaria di trasformazione, il secondo ne considera una a più argomenti.
Per ogni funzione binaria totale *f* esiste una funzione calcolabile *s* tale che, per ogni y:

$$\phi_{f(s(y),y)} = \phi_{s(y)}$$

Dimostrazione:
Come nel primo teorema di prima, usiamo l'autoapplicazione ovunque noi vogliamo che vi sia il punto fisso.

Consideriamo una funzione *g* definita come:

$$g(x,y,z) = \phi_{f(\phi_x(x),y)}(z)$$

Per smn (applicato due volte) abbiamo che:

$$\phi_{\phi_{r(y)}(x)}(z) = \phi_{h(x,y)}(z) = g(x,y,z)$$

Posto $s(y) = \varphi_{r(y)}(r(y))$, per un qualunque z:

$$\phi_{s(y)}(z) = \phi_{\phi_{r(y)}(r(y))}(z) = \phi_{h(r(y),y)}(z) = g(r(y),y,z) = \phi_{f(\phi_{r(y)}(r(y)),y)}(z) = \phi_{f(s(y),y)}(z)$$

---

44) Cosa si intende per riducibilità?
Siano A,B $\subseteq$N. A si dice riducibile a B ($A \leq_m B$) se esiste una funzione f totale calcolabile tale che

$$x \in A \Leftrightarrow f(x) \in B$$

Inoltre, due insiemi si dicono equivalenti ($=_m$) se sono mutualmente riducibili.

La relazione di riducibilità è riflessiva e transitiva.

---

45) Quando un insieme si dice m-completo?
Un insieme A si dice m-completo se è r.e. ed ogni insieme B r.e. è riducibile ad esso:

$$\forall B \text{ tale che r.e., } B \leq_m A$$

---

46) Quando un insieme è creativo? Quando è produttivo?
Un insieme A si dice ***produttivo*** se esiste una ***f*** totale calcolabile tale per cui per ogni ***i*** abbiamo:

$$W_i \subseteq A \rightarrow f(i) \in A/W_i$$

ovvero, se W_i (ovvero il dominio di convergenza della funzione *i*) è sott'insieme di A, allora *f*(i) è contenuto in $A - W_i$. Notare come Wi sia r.e., mentre A non è r.e.

Un insieme produttivo NON può essere r.e., altrimenti $A/Wi = \varnothing$.

Un insieme A si dice **creativo** se è r.e. e il suo complementare è produttivo.

---

47) Cosa si intende per insiemi immuni e semplici?
A si dice immune se è infinito e nessun suo sott'insieme infinito è r.e., altrimenti si dice se è r.e. e !A è immune.

---

48) Dimostrare che l'insieme K è creativo.

Siccome sappiamo già che l'insieme K è r.e., dobbiamo solo dimostrare che l'insieme !K sia produttivo.

Dobbiamo quindi dimostrare che esiste una *f* totale calcolabile tale per cui, per ogni ***i*** :

$$Wi \subseteq !K \rightarrow i \in !K/Wi$$

Abbiamo scelto come f la funzione identità.

Vogliamo dimostrare che $\mathbf{i \in !K \wedge i \notin Wi}$ (quindi che!K/Wi non sia vuoto).

1) Per dimostrare che i ∈ !K, supponiamo per assurdo che i ∈ K (ovvero all'insieme delle funzioni convergenti). Allora i ∈ Wi (ovvero al dominio di convergenza di ϕ_i), ma Wi ⊆ !K (per ipotesi), quindi i ∈ !K.

2) Dobbiamo dimostrare che **i ∉ Wi**. Se i appartenesse a Wi, allora apparterrebbe anche a K, ma abbiamo appena visto che non è così.

Dunque, K è creativo.

---

49) Dimostrare il teorema di Rice usando il teorema del punto fisso.

Il teorema di rice ci dice che una proprietà estensionale è ricorsiva (quindi decidibile) solo se banale, quindi se sempre vera o sempre falsa.

Supponiamo che A sia ricorsivo non banale, ovvero esistono due i,j tali che i ∈ A e j ∉ A.
Essendo A ricorsivo, possiamo costruire una funzione h definita come:

$$h(x) = \begin{cases} i & se\ x \in \overline{A} \\ j & se\ x \in A \end{cases}$$

Dunque

$$h(x) \in A \iff x \in \overline{A}$$

Per il teorema del punto fisso, siccome *h* è una funzione totale calcolabile, esiste un indice ***b*** tale per cui $\phi_{h(b)} = \phi_b$. Ma allora arriviamo ad un assurdo, infatti:

$$b \in A \Leftrightarrow h(b) \in A \Leftrightarrow b \in \overline{A}$$

Questo perché A è estensionale, quindi b ∈ A e h(b) ∈ A

che è un assurdo.

Seconda parte

1A) Quando un problema è in P? Quando è in NP?
Appartengono nella classe di complessità P tutti i problemi che possono essere risolti in tempo polinomiale.

Con NP si intendono i problemi di classe di complessità polinomiale non deterministica. Tutti i problemi in NP sono problemi di facile verifica, ovvero per cui possiamo .
Esempi di problemi in NP sono il commesso viaggiatore, il problema dello zaino..

2A) Teorema di NP come proiezione di P.

Un insieme A (o un problema A) è in NP se e solo se esiste un insieme B (di certificati per A) tali per cui l'appartenenza di una coppia $(x, c_x) \in B$ sia **decidibile in tempo polinomiale**. Inoltre deve esistere un polinomio p tale che dia un upper bound alla dimensione (in spazio) del certificato.

$$x \in A <=> \exists c_x, |c_x| \leq p(|x|) \land (x, c_x) \in B$$

3A) NP vs coNP (intersezione di NP e coNP).

L'insieme NP rappresenta la classe di tutti quei problemi per cui si può verificare in tempo polinomiale che le istanze dei problemi per cui la risposta è "si".

L'**insieme coNP**, invece, rappresenta la classe di tutti quei problemi per cui si può verificare in tempo polinomiale le istanze dei problemi per cui la risposta è "no".
Ad esempio, possiamo prendere il problema di tautologicità TAU, in cui basta trovare una assegnazione per cui la formula sia falsa per dimostrare che non è dentro TAU.

Per ogni insieme in NP, ne esiste uno in coNP, ovvero

$$A \in coNP <=> !A \in NP$$

Non si sa se NP != coNP.
P è contenuno nell'intersezione fra questi due insiemi.

4A) Cosa si intende con problema della soddisfacibilità?
Il problema della soddisfacibilità (SAT) chiede se, data una formula proposizionale, si può determinare se questa è soddisfacibile, cioè se esiste una attribuzione di valori di verità alle variabili proposizionali che rende vera la formula.

5A) Cosa si intende con DTIME e DSPACE?

Una **classe di complessità** rappresenta un insieme di problemi di una certa complessità.

DTIME e DSPACE sono le classi deterministiche di complessità, e le definiamo in questo modo.

Data una funzione $f:N \rightarrow N$, si introducono le seguenti classi di complessità:

- `DTIME(f)` $= \{L \subseteq \Sigma^* : \exists M.\ L = L_M \land t_M \in O(f)\}$, ovvero l'insieme di quei linguaggi tali per cui esiste una macchina M che li riconosce e tali per cui t_M cresce al più quanto f. In altre parole, l'insieme dei problemi decidibili in tempo *f*.
- `DSPACE(f)` $= \{L \subseteq \Sigma^* : \exists M.\ L = L_M \land s_M \in O(f)\}$, ovvero l'insieme dei problemi decibili in spazio f.

Dove,

- $t_M(n)$ rappresenta il tempo massimo di computazione per un input di dimensione n. (Quindi, rappresenta la complessità in tempo di una MdT).
- $s_M(n)$ rappresenta lo spazio massimo di computazione per un input di dimensione n. (Quindi, rappresenta la complessità in spazio della MdT).

6A) Teorema tempo-spazio. Che cosa ne consegue?

Per ogni $f : N \to N$ si ha:

$$\mathrm{DTIME}(f) \subseteq \mathrm{DSPACE}(f) \subseteq \bigcup_{c \in N} \mathrm{DTIME}(2^{c(log+f)})$$

Ne consegue che una computazione che richiede spazio *f* non può richiedere tempo maggiore di 2^f (a meno di una costante).

7A) Spiegare come mai otteniamo il risultato del teorema tempo-spazio.

Prima di tutto, il punto fondamentale del teorema tempo-spazio deriva dal fatto che per occupare nuovo spazio, c'è bisogno di tempo.
Se un linguaggio viene riconosciuto in tempo *f*, quindi, allora sarà riconoscibile in spazio *f*, siccome non si potrà occupare più di quello spazio. Tuttavia, ci sono altre cose che dobbiamo considerare.

Se siamo passati per tutti gli stati interni della MdT, e abbiamo scritto la memoria in tutti i modi possibili, allora la computazione DEVE terminare, **in quanto questo ne presenta un limite**.
Altrimenti, ci troveremo in un loop infinito.

Supponiamo quindi che n sia la lunghezza della stringa massima. Il numero di modi in cui possiamo scrivere la stringa di lunghezza n è $|\Gamma|^n$, dove Γ è il nostro alfabeto. Il numero di stati possibili è |Q|.
Considerando che abbiamo k nastri, il numero delle configurazioni è pari a:

$$t(n) \leq |Q| \cdot k \cdot |\Gamma|^{s_M(n)} \qquad \text{(limite al tempo)}$$

Dove $s_M(n)$ è la quantità massima di celle usate dai nastri in quel momento. Dunque, quanto detto, ci dice che il tempo richiesto per una computazione su un input n è sicuramente minore di quella quantità.

Dobbiamo ora trasformare quanto appena detto in base 2.
Per farlo, sappiamo che la macchina richiedeva uno spazio $s_M(n)$. per cui lavora con complessità *f*, a meno di una costante *c*. Inoltre, per il nastro in input, ci interessa sapere esattamente dove è arrivata la testina su questo nastro (e abbiamo n possibilità, dove n è la lunghezza della stringa in input).
Otteniamo così:

$$2^{c(\log(n)+f(n))+c}$$

Dove:

- $c \in N$ è una costante
- f(n) lo aggiungiamo siccome lo spazio cresce in funzione di O(f), dunque posso rimpiazzare $s_M(n)$ con f(n)
- log(n) ci serve per il cambio di base.

8A) Quando una funzione si dice costruibile in tempo? Quando una funzione è costruibile in spazio? Che lemma lega queste due cose?

Una funzione f: N → N si dice **costruibile in tempo** se esiste una MdT M sull'alfabeto Σ = {0} che calcola una funzione che effettua una trasformazione $0^n \vdash 0^{f(n)}$ in tempo $t_M \in O(f)$.

Analogamente, f è costruibile in spazio se valgono le stesse condizioni con s_M al posto di t_M.

[Essenzialmente, si traforma in una stringa di 0 lunga n in una lunga f(n).]

Il lemma che lega queste due cose è che:
Ogni funzione costruibile in tempo è anche costruibile in spazio, ma non viceversa.

La funzione logaritmo è costruibile in tempo, ma non in spazio.

9A) Teorema della gerarchia in spazio. Spiegare e dimostrare.

Teorema:
Data una funzione ***s* costruibile** in spazio e $\log \in O(s)$ (ovvero s cresce come il logaritmo), allora vale che:

$$O(s) \subseteq O(s') \Leftrightarrow DSPACE(s) \subseteq DSPACE(s')$$

Dimostrazione:
=>) Questa parte è ovvia, siccome se O(s) ⊆ O(s'), io posso risolverse un problema che usa spazio *s* usando spazio *s*'.

<=) Vogliamo dimostrare che aumentando l'ordine di grandezza delle risorse allora riusciamo a risolvere problemi che prima non eravamo in grado di risolvere. Per dimostrarlo, ci basta dimostrare:

$$O(s) \not\subseteq O(s') \Rightarrow DSPACE(s) \not\subseteq DSPACE(s')$$

La prima parte della formula equivale a dire che $s \notin O(s')$, ovvero che s cresce più di s' asintoticamente, ovvero che per ogni $f \in O(s')$, esistono infiniti n tali per cui s(n) > f(n).
Sapendo ciò, dobbiamo dimostrare che:

$$DSPACE(s) \not\subseteq DSPACE(s')$$

ovvero che esista un linguaggio L tale che è riconoscibili in spazio ***s*** ma non in spazio ***s'***.
Dobbiamo trovare questo linguaggio L.

Definiamo L come un insieme di codici $\lceil M \rceil$ di MdT tali che:

$$L := \{ \lceil M \rceil \mid \underbrace{space_M(|\lceil M \rceil|) \leq s(|\lceil M \rceil|)}_{\text{complessità}} \text{ e } \underbrace{\lceil M \rceil \notin L_M}_{\text{diagonalizzazione}} \}$$

Dove:

- La prima condizione garantisce che lo spazio richiesto dalla macchina sia limitato da s, quindi che $L \in DSPACE(s)$ (che però dobbiamo ancora dimostrare formalmente).
- $\lceil M \rceil$ è dunque l'insieme delle quintuple che definisce il codice della macchina, dunque L è dunque un linguaggio di codici. La seconda condizione dice che stiamo prendendo le macchine che non riconoscono il proprio codice (diagonalizzazione).

Supponiamo di avere una macchina M_0 che riconosca L ($L=L_{M0}$).
Se $\lceil M_0 \rceil \in L$, allora $\lceil M_0 \rceil \notin L_{M0}$ che è una contraddizione visto che $L=L_{M0}$.
Dunque, $\lceil M_0 \rceil \notin L=L_{M0}$ (abbiamo rispettato il vincolo di diagonalizzazione).
Adesso vogliamo arrivare ad una contraddizione (ovvero che $\lceil M_0 \rceil \notin L \rightarrow \lceil M_0 \rceil \in L$) e concludere che $L \notin DSPACE(s')$.

Supponiamo allora che $s_{M0} \in O(s')$, ovvero che M_0 lavori in spazio s'. In questo modo, però, non rispettiamo il vincolo di complessità:

$$s_{M0}(|\lceil M_0 \rceil|) \leq s(|\lceil M_0 \rceil|)$$

Siccome sappiamo che **s cresce più di s'** asintoticamente, **ma non sappiamo se effettivamente questo è vero per quel determinato input** (e quindi non asintoticamente).
In questo modo però non abbiamo la contraddizione desiderata.

Complichiamo allora la definizione del linguaggio L, facendo **padding** dei codici con infinite stringhe *x*, per catturarne il comportamento asintotico. Otteniamo:

$$L := \{ \langle \lceil M \rceil, x \rangle \mid \underbrace{space_M(|\langle \lceil M \rceil, x \rangle|) \leq s(|\langle \lceil M \rceil, x \rangle|)}_{\text{complessità}} \text{ e } \underbrace{\langle \lceil M \rceil, x \rangle \notin L_M}_{\text{diagonalizzazione}} \}$$

Supponiamo allora che $L \in DSPACE(s')$, per cui esiste una macchina M che riconosce L in spazio O(s').

Per questo, abbiamo infiniti n per cui $s(n) > s_{M0}(n)$ (visto che appartiene ad O(s')).
Scegliamo un padding x_0 qualsiasi per cui valga questa proprietà. Per questo vale:

$$\begin{aligned} \langle \lceil M_0 \rceil, x_0 \rangle \in L &\Leftrightarrow space_{M_0}(|\langle \lceil M_0 \rceil, x_0 \rangle|) \leq s(|\langle \lceil M_0 \rceil, x_0 \rangle|) \text{ e } \langle \lceil M_0 \rceil, x_0 \rangle \notin L_{M_0} \\ &\Leftrightarrow \langle \lceil M_0 \rceil, x_0 \rangle \notin L_{M_0} \end{aligned}$$

In questo caso, siamo riusciti ad arrivare alla contraddizione (siccome sicuramente esisterà una stringa x_0 che riesce a catturare il comportamento asintotico della formula).

Adesso, dobbiamo dimostrare che $L \in DSPACE(s)$ come avevamo promesso prima. Per farlo, dobbiamo esibire l'esistenza di una MdT che riconosce L e lavora in spazio O(s).

La nostra macchina dunque dovrà soddisfare due vincoli:

- Per il vincolo di complessità, bisogna salvarsi su un nastro il valore $s(|\langle \lceil M \rceil, x \rangle|)$, che è lo spazio massimo che può usare la macchina. Se la simulazione cerca di superare questo bound, viene prontamente arrestata e la stringa non appartiene al linguaggio.

- Applichiamo il teorema tempo-spazio per ottenere un bound anche al tempo, ed evitare dei casi di *divergenza*, e dovremo memorizzarlo.
  Questo timeout ha valore $t = |Q| \cdot 2 \cdot |\Sigma|^{s(n)} \cdot |y|$, dove |y| è la dimensione dell'input, che dobbiamo trasformarlo in binario con complessità log(t).
  Le uniche parti non costanti di questa formula sono y e s(n), quindi la complessità in spazio per calcolarlo è $O(t) = O(\log(y)+s(n)) = O(s(n))$. s(n) assorbe il logarimo, siccome nel teorema abbiamo supposto che $\log \in O(s)$.

Con questi due vincoli, la nostra simulazione non consuma più spazio di *s*, per cui $L \in DSPACE(s)$.
La macchina accetta l'input y se la simulazione di M termina rifiutandolo, oppure se si è esaurito il tempo, ma non lo spazio.

10A) Teorema della gerarchia in tempo. Come cambia la dimostrazione rispetto al teorema della gerarchia in spazio?

Sia t una funzione costruibile in tempo e $n \in O(t)$. Allora:

$$O(t) \subseteq O(t') \Rightarrow DTIME(t) \subseteq DTIME(t') \Rightarrow O(\frac{t}{\log t}) \subseteq O(t')$$

La dimostrazione cambia in questo modo:

- Il vincolo di spazio viene omesso, e si usa solo un vincolo di tempo che viene calcolato in funzione dell'input. Questo implica che ad ogni passo della simulazione della macchina dovremo aggiornare il timeout. Questa operazione ha un costo costante, approssimabile a *log(t)*. Per compensare a questa aggiunta, facciamo la divisione per log(t).
- Il linguaggio L ora viene definito in questo modo:

$$L := \{(\lceil M, x \rceil) \mid t_M(|(\lceil M \rceil, x)|) \leq \frac{t(|(\lceil M \rceil, x)|)}{log\ t(|(\lceil M \rceil, x)|)} \ \wedge \ (\lceil M \rceil, x) \notin L_M\}$$

11A) Cosa si intende con padding? A cosa serve?

Quando parliamo di **padding** (o linguaggio di padding), intendiamo un linguaggio esteso con dei caratteri speciali ridondanti. Il padding viene usato per dimostrare le proprietà dei linguaggi, e per dimostrare, ad esempio, che alcune classi sono diverse.

12A) Dimostrare che EXP != PSPACE

1. Prima di tutto vogliamo dimostrare che EXP $\subset$DTIME($2^{n\wedge 2}$).
   Per il teorema della gerarchia, sappiamo che:
   $$O(2^{cn}) \subset O(2^{n^2}), \text{ allora } DTIME(2^{cn}) \subset DTIME(2^{n^2})$$
   Adesso ci chiediamo se l'inclusione stretta valga anche per EXP = $\bigcup_{c \in \mathcal{N}} DTIME(2^{cn})$.
   Tuttavia, possiamo subito dire che la cosa non è ovvia, siccome non è detto che possiamo contenere ***strettamente*** l'unione per *c* di tutti gli insiemi DTIME(2^{cn}), anche se ciascuna singola classe è contenuta.
   Consideriamo allora:
   $$\bigcup_{c \in \mathcal{N}} DTIME(2^{cn}) \subseteq DTIME(2^{n^\alpha}) \subset DTIME(2^{n^2})$$
   dove $1 < \alpha < 2$. Siccome tutte le singole classi DTIME(2^{cn}) sono strettamente contenute in $DTIME(2^{n^\alpha})$, possiamo dire che anche la loro unione sia contenuta in esso in modo stretto o uguale a $DTIME(2^{n^\alpha})$. Il fatto che la l'inclusione stretta non venga preservata non ci dà fastidio, siccome comunque sappiamo che $DTIME(2^{n^\alpha}) \subset DTIME(2^{n^2})$. Dunque, sicuramente $EXP \subset DTIME(2^{n^2})$.
2. Supponiamo che EXP=PSPACE, e sia un generico $L \in DTIME(2^{n^2})$.
   Eseguiamo il padding quadratico di L, quindi:
   $$L' = \{w\#^l \mid l = |w|^2 - |w|, w \in L\}$$
   Sappiamo che L' ha una complessità inferiore a L, siccome w ha una dimensione che è una radice della stringa completa, quindi L' ha complessità $O(2^n)$, e quindi **L' $\in$ EXP**.
   Siccome abbiamo supposto che EXP = PSPACE, allora **L' $\in$ PSPACE**, dunque esiste un k tale per cui L' $\in$ DSPACE(n^k).

Supponiamo a questo punto di avere una MdT M' che riconosce L', e che vogliamo utlizzare per ***costruire*** un'altra MdT M che riconosca L, ovvero che determini se $w \in L$.
M agisce in questo modo:

Fa il padding di w ottenendo $w\#^{l}$, dopodiché consegna la stringa paddata a M'.
Il padding ha costo $O(n^2)$, e la stringa risultante ha dimensione n^2. M' lavora in spazio n^k, quindi il costo in spazio sarà $O(n^{2k})$, da cui deriviamo che $L \in DSPACE(n^{2k})$.
Questo significa che $L \in PSPACE$.

In base alla supposizione che EXP=PSPACE, però, **qualunque** $L \in DTIME(2^{n^2})$**, sta anche in EXP, usando questa costruzione**.
Prima però abbiamo dimostrato che $EXP \subset DTIME(2^{n^2})$, ovvero che ci sono dei linguaggi L che stanno in $DTIME(2^{n\wedge 2})$, ma non in EXP, il che porta ad una contraddizione.

13A) Classi di complessità non deterministiche
Le classi di complessità non deterministiche sono:
Data una funzione $f:N \to N$, si introducono le seguenti classi di complessità:

- `NTIME(f)` $= \{L \subseteq \Sigma^* : \exists M.\ L = L_M \wedge t_M \in O(f)\}$, ovvero l'insieme di quei linguaggi tali per cui esiste una MdTN M che li riconosce e in tempo O(f).
- `NSPACE(f)` $= \{L \subseteq \Sigma^* : \exists M.\ L = L_M \wedge s_M \in O(f)\}$, ovvero l'insieme di quei linguaggi tali per cui esiste una MdTN M che li riconosce e in spazio O(f).

Altre classi che possiamo definire:

- $NP := \bigcup_{c \in N} NTIME(n^c)$
- $NEXP := \bigcup_{c \in N} NTIME(2^{cn})$
- $NLOGSPACE := NSPACE(log)$
- $NPSPACE := \bigcup_{c \in N} NSPACE(n^c)$

14A) Simulazione del nondeterminismo, primo teorema.

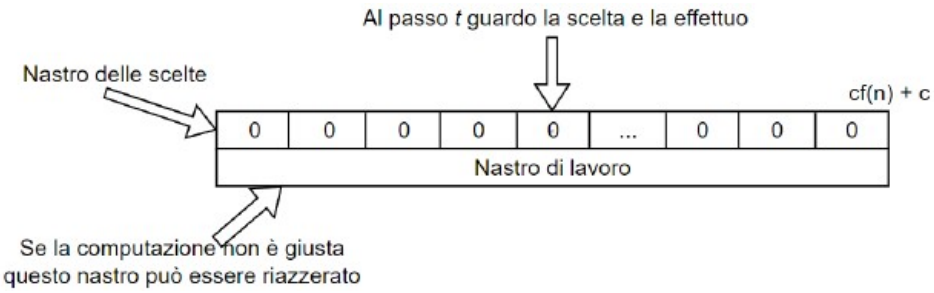
Teorema:
Per ogni $f:N \to N$ costruibile in spazio:

$$NTIME(f) \subseteq DSPACE(f)$$

Dimostrazione:

- Supponiamo di avere una MdTN M che riconosce un linguaggio L (ovvero riconosce ogni stringa del linguaggio) in tempo O(*f*).
- Costruiamo ora una macchina MdT deterministica M' a due nastri: uno è il nastro di lavoro, mentre l'altro contiene le possibili scelte da effettuare. Per semplicità, supponiamo che M abbia unicamente due possibili scelte (codificabili con 0 e 1).
- Supponiamo che l'input x abbia dimensione n, e codifichiamo con i simboli {0,1} la scelta che la macchina deterministica effettua ad ogni passo (ad es. 0 fai la prima scelta, 1 la seconda).
- Possiamo inizialmente supporre di inizializzare i nostri nastri in modo da avere tutti 0 nel primo nastro, e x scritto sul nastro di lavoro.
- Al passo t-esimo, la macchina guarda il primo nastro, e in base al suo stato decide che scelta prendere. Dopodiché simula la computazione sul secondo nastro.
- Se la computazione non porta a nulla, possiamo riazzerare il nostro nastro di lavoro, riportandolo allo stato iniziale copiando x sopra di esso, e si cambia una delle possibili scelte del nastro delle scelte.

- Siccome ad ogni passo il nastro può essere riutilizzato, la macchina non consuma uno spazio superiore ad *f*.



15A) Simulazione del nondeterminismo, secondo teorema.

Teorema:
Per ogni f: N → N costruibile in spazio:

$$NSPACE(f) \subseteq DTIME(2^{c(\log+f)})$$

con c ∈ N.

Dimostrazione:
Consideriamo le configurazioni possibili che ci permettono di riconoscere una stringa in input.
In questo caso, siccome usiamo una MdTN, abbiamo un grafo di configurazioni, che parte da uno stato iniziale ed arriva a tanti possibili stati finali.

Per fare una simulazione deterministica, dunque, **sarà necessario esplorare il grafo alla ricerca di una configurazione di accettazione**.
Siccome sappiamo che la nostra macchina MdTN lavora in spazio f, possiamo eliminare tutte quelle configurazioni che portano ad un'occupazione in spazio maggiore di f.

Se riusciamo a trovare il numero dei nodi (ciascuno dei quali corrisponde ad una configurazione della MdTN) del nostro grafo, avremmo un bound del tempo richiesto per l'esplorazione. Questo è dato dal numero totale delle configurazioni possibili, che è lo stesso della MdT, e che è pari a:

$$n \cdot |Q| \cdot k \cdot |\Sigma|^{cf(n)+c}$$

dove:

- *n* sono le possibili posizioni in cui si trova la testina sul nastro di input.
- k è il numero dei nastri di lavoro
- $|\Sigma|^{cf(n)+c}$ è il numero dei modi in cui possiamo scrivere i nastri on un alfabeto di dimensione $|\Sigma|$

Adesso possiamo portare tutto in potenza di base 2, osservando che i termini più significativi della formula precedente sono in particolare il primo e l'ultimo, ottenendo (attraverso il cambio di base)

$$2^{c'(\log+f)}$$

dove *c'* è un valore opportuno.
Il costo dell'esplorazione in ampiezza di un grafo con questo numero di nodi è in $DTIME(2^{c'(\log+f)})$.

I due teoremi della simulazione ci dicono come se vogliamo simulare una MdTN con spazio/tempo *f*, allora la sua simulazione deterministica richiederebbe spazio/tempo 2^f.

16A) Algoritmo di raggiungibilità in spazio $\log^2(n)$.
Mentre negli algoritmi classici dobbiamo **memorizzare** tutti i nodi in apposite strutture dati (e la dimensione di ogni nodo è di log(n) mentre la complessità in spazio è di nlog(n)), con questo algoritmo possiamo determinare se due nodi u,v sono connessi usando un approccio *divide et impera*.

Algoritmo 3 Raggiungibilità II

```
procedure reachable(i, u, v)
    if i == 0 then                                   ▷ Cammino di lunghezza 1
        return u = v ∨ (u, v) ∈ E
    else ∃z . reachable(i − 1, u, z) ∧ reachable(i − 1, z, v)   ▷ Restituisce True al primo z.
    end if
    return False
end procedure
```

Osserviamo che:

- Nel caso base, in cui i=0, i cammini sono di lunghezza al più 1. Quindi due nodi sono raggiungibili in questo caso o se coincidono oppure se esiste un arco che li collega.
- Nel caso induttivo, cerchiamo un z tra tutti i nodi (con un ciclo For, che viene "simulato" dall'operatore esistenziale), in modo che sia il "punto medio" tra i due nodi da collegare. Quindi cerchiamo due cammini di lunghezza pari o inferiore del tipo u → … → z → … → v.

Il costo in spazio dell'algoritmo è dato dal numero di record d'attivazione e dalla dimensione dimensione di ogni record. Tuttavia, osserviamo che la seconda chiamata ricorsiva può riutilizzare lo spazio della prima. Infatti, quando la prima chiamata ricorsiva conclude, non ci resta che memorizzare il risultato finale.
Il numero di chiamate ricorsive annidate è al più *i* (per un cammino semplice, non ciclo). La lunghezza massima di un cammino (senza cicli) è n (ovvero il numero totale dei nodi).
Dunque i = log(n), e 2^i sarà quindi la lunghezza massima del nostro cammino.
Allora, il numero massimo delle chiamate annidate è *log(n)* (siccome *i=log(n)*), ogni record di attivazione deve memorizzare memorizzare i nodi del grafo (che sono i parametri passati in procedura). Questo ha il costo in spazio **$\log^2(n)$**.
In tempo, invece, ha complessità esponenziale.

17A) Teorema di Savitch.
Sia s : N → N una funzione costruibile in spazio, tale che *log* ∈ O(*s*). Allora:

$$NSPACE(s) \subseteq DSPACE(s^2)$$

Il teorema ci dice che possiamo simulare una macchina non deterministica che lavora in spazio s con una macchina deterministica che lavora in spazio s^2

Dimostrazione:
Possiamo pensare che la macchina non deterministica abbia un unica configurazione di accettazione. Dunque, dobbiamo controllare se esiste un cammino in questo grafo tra la configurazione di accettazione e la configurazione iniziale.

Dalla dimostrazione precedente, sappiamo che una MdTN che lavora in spazio s avrà un grafo con un numero di configurazioni al più esponenziale rispetto a s, in particolare sappiamo che il numero di nodi è limitato a $2^{cs(n)+c}$. Utilizzando **l'algoritmo di raggiungibilità che ha costo in spazio $\log^2(n)$**, calcoliamo $log(2^{cs(n)+c})$ otteniamo così che il costo è di $O(s^2)$.

18A) Conseguenze del teorema di Savitch

- PSPACE = NPSPACE, siccome sappiamo che $DSPACE(s) \subseteq NPSPACE(s) \subseteq DSPACE(s^2)$, e visto che il peggioramento è polinomiale, allora le classi si equivalgono.
- NPSPACE = coNSPACE, siccome le classi deterministiche sono chiuse rispetto alla complementazione.

Questi risultati non valgono anche per il tempo, altrimenti riusciremmo a dimostrare che **P = NP**.

19A) Teorema della proiezione (complessità)

Dato un linguaggio $A \subseteq \Sigma^*$, allora $A \in NP$ se e solo se esiste un linguaggio di coppie $B \in P$ ed un polinomio *p* tale che, per ogni $x \in \Sigma^*$.

(1) $$x \in A \iff \exists y.|y| \leq p(|x|) \ \wedge (x,y) \in B$$

Dimostrazione:
→) Supponiamo che $A \in NP$. Vogliamo allora dimostrare che esiste un linguaggio di coppie $B \in P$ e un polinomio *p* tale che per ogni $x \in \Sigma^*$, (1) è vero.

Siccome A sta in NP, sappiamo che esiste una MdTN che riconosce A. Inoltre, consideriamo il linguaggio B fatto dalle coppie (stringa, certificato), e prendiamo come certificato l'insieme delle scelte fatte dalla MdTN per riconoscere A. Il numero di scelte è al più polinomiale, siccome la MdTN sta in NP, quindi abbiamo un bound al numero di scelte che sono state fatte (che ci garantisce che la forma).

A questo punto possiamo simulare la macchina in maniera deterministica. Alla fine della computazione, controlliamo se lo stato è di accettazione o meno. Se la stringa non sta nel linguaggio, allora la computazione non esiste e quindi non esiste il certificato.

←) Supponiamo che esista un linguaggio $B \in P$ nella forma $(x, y) \in B$. Sia A la proiezione esistenziale di B, con (1) vero. Vogliamo dimostrare che $A \in NP$.

Dobbiamo quindi trovare una MdTN che lavori in tempo polinomiale e riconosca A.
Questa macchina prende x in input e **non deterministicamente** inventa il certificato y, per poi dare la coppia (x, y) alla macchina deterministica che riconosce B. Generare il certificato ha un costo al più polinomiale, siccome il certificato ha dimensione al più polinomiale (per ipotesi $|y| \leq p(|x|)$).

La macchina appena costruita riconosce x (se sta in A), siccome se $x \notin A$ per ipotesi il certificato esiste.

Il costo di questa macchina appena definita è quindi è polinomiale, siccome la macchina deterministica che sfrutta è polinimiale e genera non deterministicamente il certificato, e appunto riconosce A, quindi $A \in NP$.

20A) Riducibilità nella complessità
Diciamo che il problema A è riducibile al problema B se esiste un algoritmo per risolvere il problema B che può essere utilizzato per risolvere il problema A.

Formalmente,
Dati due linguaggi $A,B \in \Sigma^*$, allora:

- A è riducibile **in tempo polinomiale** a B ($A \leq_P B$) se esiste una funzione $f \in FP$ tale per cui per ogni $x \in \Sigma^*$,

$$x \in A <=> f(x) \in B$$

- A è riducibile in spazio logaritmico a B ($A \leq_L B$) se esiste una funzione $f \in$ FLOGSPACE tale per cui per ogni $x \in \Sigma^*$,

$$x \in A \iff f(x) \in B$$

Per capire cosa si intende con FP e FLOGSPACE, dobbiamo introdurre le classi di complessità per le funzioni:

- FTIME(t) := $\{f : \Sigma^* \to \Sigma^* \mid \exists M\ f = f_M \wedge t_M \in O(t)\}$, ovvero l'insieme di quelle funzioni di trasformazioni dei linguaggi per cui esiste una MdT che le calcola in tempo t.
- FSPACE(t) := $\{f : \Sigma^* \to \Sigma^* \mid \exists M\ f = f_M \wedge s_M \in O(t)\}$, ovvero l'insieme di quelle funzioni di trasformazioni dei linguaggi per cui esiste una MdT che le calcola in spazio t.

FP è l'unione di tutte le classi FTIME(n^c), mentre FLOGSPACE è FSPACE(log).

21A) Chiusura per riducibilità
Supposto che ≤ è un preordine, allora una classe di linguaggi C è chiusa rispetto a ≤ se

$$A \leq B \wedge B \in C \to A \in C$$

Ovvero, se B è nella nostra classe di linugaggi, tutti i linguaggi riducibili a B stanno nella nostra classe.

22A) Quando un problema è arduo? Quando è completo?
Data una classe di linguaggi *C*, un preordine ≤ e un linguaggio B, diciamo che:

- B è ***C*-arduo** rispetto a ≤, se per ogni $A \in C$ vale $A \leq B$
- B è ***C*-completo** rispetto a ≤, se oltre ad essere C-arduo vale anche che $B \in C$.

23A) Cosa succederebbe se per qualche problema NP-completo A valesse che $A \in P$?

Teorema:
Se per qualche problema NP-completo A vale che $A \in P$, allora P=NP.

Dimostrazione:
Dato un qualunque $B \in NP$, visto che la trasformazione avviene in tempo polinomiale (e quindi è un **preordine**), se $A \leq_P B$ e $A \in P$, allora $B \in P$.
Quindi abbiamo che $NP \subseteq P$, ma siccome sappiamo che $P \subseteq NP$, allora P=NP.

24A) Dimostrare che se P=NP, allora ogni problema non banale $A \in NP$ è NP-completo. (che vuol dire che per dimostrare che P!=NP, allora basta dimostrare che un problema non banale in NP non sia completo)

Dimostrazione: Questo lemma vale siccome tutti i problemi in P sono P-completi, e dobbiamo dimostrare questo.
Supponiamo di avere due insiemi A, $B \in P$. Questo significa che la funzione caratteristica di B è calcolabile in tempo polinomiale. Modifichiamo la funzione caratteristica di B facendo in modo che:

- Se $x \in B$, restituisco $a_0 \in A$
- Se $x \notin B$, restituisco $a_1 \notin A$

In questo modo, abbiamo trasformato la funzione caratteristica di B in una funzione di riduzione, che restituisce degli elementi in A, quindi $B \leq_P A$ e qualunque coppia di problemi polinomiali non banali sono riducibili l'uno all'altro.

Se poi NP = P, allora ogni problema non banale in NP sarebbe completo.

Ne consegue che, se io voglio dimostrare che P != NP, allora basta dimostrare che esiste un problema in NP non banale che NON è completo.

25A) Teorema di Cook. Dimostralo.

Teorema: SAT è NP-completo.

Dimostrazione:

- Prima di tutto, dimostrare che SAT $\in$ NP è semplice, siccome possiamo verificare in tempo lineare una formula proposizionale, una volta che abbiamo (come certificato) i valori associabili alle variabili.
- Dunque, **dobbiamo dimostrare che il problema è NP-arduo**. Ovvero, dobbiamo dimostrare che possiamo ridurre in tempo polinomiale un qualunque linguaggio L $\in$ NP a SAT. (Dire che L $\in$ NP significa che esiste una MdTN M che riconosce L in tempo polinomiale).
- Sia L $\in$ NP, dobbiamo quindi dimostrare che:

$$\forall x \in \Sigma^*.\ x \in L \Leftrightarrow \psi x \in SAT$$

- Dobbiamo quindi definire una formula ψx che dipende da x. Se x $\in$ L, sappiamo che esiste almeno una computazione che porta al riconoscimento di x al più in *p(n)* passi. Sappiamo inoltre che per il teorema tempo-spazio, l'occupazione del nastro non eccede p(n).
  ψx deve descrivere la computazione di M per il riconoscimento di x.
- Possiamo rappresentare la computazione dell'input x come una **matrice** di dimensione p(n)*(p(n)+2) [queste 2 colonne in più delimitano l'inizio e la fine del nastro].
  - Ogni riga di questa matrice deve rappresentare la configurazione in un particolare passo della computazione
  - ogni colonna rappresenta una cella del nastro.
- Inoltre, siccome le righe devono descrivere la computazione, bisogna sapere lo stato interno della macchina e la posizione della testina.
  In corrispondenza della posizione della testina quindi scriviamo la coppia (q, *a*), dove q è lo stato attuale della MdTN ed *a* è il carattere in quella posizione.
- A sto punto, definiamo delle variabili proposizionali $y_{i,j,a}$ tali che

$$y_{i,j,a} = \text{true} \iff \text{Il carattere j nella i-esima posizione è } a.$$

Dobbiamo ora definire dei vincoli per le variabili proposizionali, in modo da ottenere la computazione attesa:

1. Una determinata cella ci può essere uno e un solo carattere. Ovvero, "*per ogni i,j almeno una variabile proposizionale deve essere vera (per almeno un carattere); Inoltre, per tutte le coppie di caratteri a,b tali che a ≠ b, al massimo uno tra $y_{i,j,a}$ e $y_{i,j,b}$ è vero*".

$$\psi_0 := \bigwedge_{i=0}^{p(n)} \bigwedge_{j=1}^{p(n)} \left(\bigvee_{a \in \Gamma'} y_{i,j,a} \ \wedge \bigwedge_{a,b \in \Gamma',\ a \neq b} (\neg y_{i,j,a} \ \vee \ \neg y_{i,j,b}) \right)$$

2. Vincoliamo tutte le caselle nelle colonne 0 e p(n)+1 ad avere i caratteri blank (B).

$$\psi_1 := \bigwedge_{i=0}^{p(n)} y_{i,0,B} \ \wedge \ y_{i,p(n)+1,B}$$

3. Al passo iniziale, il nastro deve contenere la stringa x, mentre il resto del nastro deve essere inizializzato a blank. La testina deve inoltre essere su x1 (ovvero il primo carattere di x).

$$\psi_2 := y_{0,1,(} \underbrace{q_0}_{\text{Stato iniziale}} {}_{,x_1)} \wedge \bigwedge_{j=2}^{p(n)} y_{0,j,x_j} \wedge \underbrace{\bigwedge_{j=n+1}^{p(n)} y_{0,j,B}}_{\text{Da } n+1 \text{ tutti blank}}$$

4. Visto che la configurazione porta ad uno stato d'accettazione per x, la configurazione finale (che supponiamo si ottenga al passo t=p(n)) dev'essere in uno stato di accetazione. Dunque, sia *a* una coppia del tipo (qf, b), dove qf è uno stato finale di accettazione.

$$\psi_3 := \bigvee_{j=1}^{p(n)} \bigvee_{a \in F \times \Gamma'} y_{p(n),j,a}$$

Questo *a* si intende.

5. Ci focalizziamo ora sulla relazione di transizione. Il vincolo ci dice che "*Se al tempo i nella posizione j-1 ho* **a**, *nella posizione j ho* **b** *e nella posizione j+1 ho* **c** *e la testina non è in queste posizioni, allora in j nel passo i+1 avremo ancora b*"

$$\psi_4 := \bigwedge_{i=0}^{p(n)-1} \bigwedge_{j=0}^{p(n)} \bigwedge_{a,b,c \in \Gamma} (y_{i,j-1,a} \wedge y_{i,j,b} \wedge y_{i,j+1,c} \rightarrow y_{i+1,j,b})$$

6. Questo vincolo definisce il modo in cui avvengono i veri e propri passaggi delle configurazioni. La formula $\Delta_{q,a,i,j}$ descrive le possibili evoluzioni non deterministiche della macchina la passo i.

$$\psi_5 := \bigwedge_{i=0}^{p(n)-1} \bigwedge_{j=1}^{p(n)} \bigwedge_{(q,a) \in Q \times \Gamma} \Delta_{q,a,i,j}$$

Possiamo definire così ψx come l'AND tra questi 6 vincoli. In questo modo, la formula è soddisfacibile solo se i 6 vincoli vengono rispettati, ovvero se e solo se esiste una computazione non deterministica di M che porta all'accettazione di x.

Siccome tutte le formule hanno dimensione polinomiale, allora sono costruibili in tempo polinomiale, quindi la trasformazione avviene in tempo polinomiale.

26A) Analogie e differenze fra Calcolabilità e Complessità
Possiamo osservare una analogia tra insiemi ricorsivi e linguaggi in P, e gli insiemi RE e i linguaggi NP.

- In maniera simile a quanto accade tra P e NP, gli insiemi RE possono essere visti come la proiezione esistenziale degli insiemi ricorsivi. Ma nella complessità aggiungiamo il vincolo sulla dimensione del certificato.
- Abbiamo visto che esistono insiemi (come ad esempio K) che sono RE non ricorsivi, dunque siamo certi che RE ≠ ricorsivi.
- Sono noti degli insiemi RE non completi.
- L'intersezione fra RE e coRE è data dagli insiemi ricorsivi. In complessità, invece, NP ∩ coNP ≠ P (es. il problema della fattorizzazione sta nell'intersezione, ma non in P).

27A) Riduzione da 3-SAT a VC (hint: archi vengono coperti)

Con VC intendiamo il problema del ricoprimento, secondo cui dato un grafo G=(V, E), un

ricoprimento è un $V' \subseteq V$ tale per cui:

$$\forall (u, v) \in E, u \in V' \vee v \in V'.$$

Ovvero per ciascun arco, almeno uno dei due vertici adiacenti è contenuto nell'insieme V'.
La versione decisionale (che è quella che ci interessa) consiste nel determinare se esiste un ricoprimento di dimensione minore o uguale ad un intero *k*.

Adesso dimostriamo che VC è NP-completo. Sappiamo che $VC \in NP$, siccome possiamo facilmente verificare se un insieme è un ricoprimento di dimensione al massimo k. Dobbiamo quindi dimostrare che VC è NP-arduo facendo una da 3-SAT a VC.

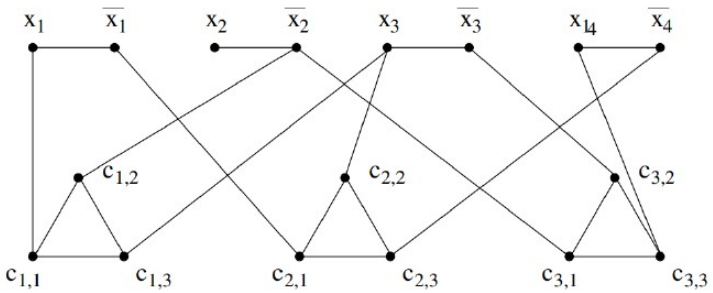
In questo modo:

- Se la formula è soddisfacibile, allora la trasformazione costruisce un grafo che ammette un ricomprimento ≤ k.
- Altrimenti, se non è soddisfacibile, non lo costruisce.

Questa trasformazione, come sempre, deve avvenire in tempo polinomiale.

Supponiamo che la formula abbiamo *m* clausole C_i, e che complessivamente abbiamo n variabili proposizionali x_i in esse. Ciò implica che $n \leq 3m$, siccome in ogni clausola abbiamo esattamente 3 letterali e le variabili possono essere ripetute.

Adesso, costruiamo un grafo G_F da 2n+3m nodi in questo modo:



Ovvero:

- Abbiamo 2n vertici xi, ¬xi tali per cui xi è connesso alla sua versione negata.
- Abbiamo 3m vertici $c_{j,k}$ (ovvero la variabile k della clausola j) che si collegano ciascuno a xi o ¬xi se Yk=xi o Yk=¬xi rispettivamente.
- Infine, $c_{j,1}$, $c_{j,2}$, $c_{j,3}$ sono connessi fra loro a triangolo.

Abbimo ottenuto G, quindi adesso dobbiamo determinare k. Consideriamo che:

- Per coprire gli archi che legano le variabili con polarità opposte, abbiamo bisogno di almeno *n* nodi.
- Per coprire ogni triangolo abbiamo bisogno di almeno 2 nodi.

Dunque, otteniamo che: k = n + 2m.

Dobbiamo ora dimostrare che se la formula sarà soddisfacibile, allora possiamo trovare in G_F un ricoprimento di dimensione almeno n + 2m (altrimenti no).

- Supponiamo che F sia soddisfacibile (ovvero che esista una attribuzione di valori di verità che renda F vera).
  Allora, per ogni clausola, c'è sempre almeno un vertice adiacente che ha valore di verità true. Dunque, colleghiamo questo nodo con uno tra i vertici di xi opposta, e scegliamo quelli per **cui il valore di verità è true**. In questo modo, abbiamo coperto tutti gli archi fra questi nodi. Per i triangoli, invece, possiamo "deselezionare" un nodo tale che il nodo

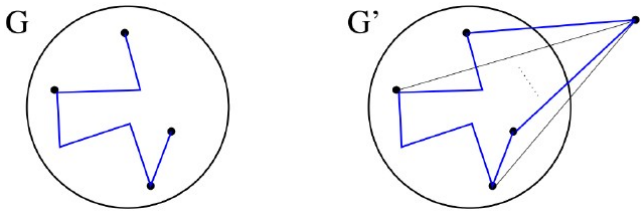
adiacente ad esso sia true. In questo modo abbiamo un ricoprimento della cardinalità che ci aspettavamo.

- Adesso dobbiamo mostrare il viceversa, ovvero che se abbiamo un ricoprimento di cardinalità n + 2m, allora F è soddisfacibile. Un ricoprimento di questa cardinalità implica che per ogni triangolo selezioniamo due vertici.
  Per completare il ricoprimento, bisogna scegliere un vertice fra i nodi associati alle variabili di polarità opposta. Se assegnamo valore true a questo vertice, allora la clausola è soddisfatta.

28A) Riduzione da Ham-P (Cammino) a Ham-C (Ciclo)

Vogliamo trasformare G in modo tale che ammetta un cammino hamiltoniano se e solo se G' ammette un ciclo hamiltoniano. Per farlo, aggiungiamo un nodo che colleghiamo con tutti i nodi in G'.

- Se G ammette un cammino hamiltoniano, allora questo diventa un ciclo hamiltoniano in G'.
- Se G' ammette un ciclo, supponiamo di avere un cammino che parta dal nuovo nodo aggiunto. Visto che andiamo a toccare tutti i nodi di G, G ammette un cammino hamiltoniano semplice.



29A) Riduzione da Ham-C (Ciclo) a Ham-P (Cammino)

Vogliamo dimostrare che G ammette un ciclo hamiltoniano se e solo se G' (ovvero il grafo trasformato) ammette un cammino hamiltoniano.

L'idea della dimostrazione è che andiamo a scegliere un nodo *i* di G e svolgiamo queste operazioni:

- Si clona il nodo scelto, creando così un nuovo nodo connesso che è collegato a tutti i vicini di *i*.
- Si aggiunge un nodo *start* collegato solo ad *i*.
- Si aggiunge un nodo *end* collegato al clone di *i*.

Dimostriamo adesso la riduzione in entrambi i sensi.

- Supponiamo che vi sia un ciclo in G, e supponiamo che inizi da *i*. Allora, il ciclo continuerà da uno dei vicini di *i* e toccherà tutti gli altri nodi, fino a tornare in un altro dei vicini di i e richiudesi in i.
  In G', invece, supponiamo che il cammino parti da start. Da qui va verso i, e segue un percorso simile al precedente. Tuttavia, al posto che chiudersi in *i*, il cammino va verso il clone di i e poi e arriva ad end. In questo abbiamo un cammino hamiltoniano.

- Supponiamo che ci sia un cammino in G' che inizi da start. Allora passa per i, e poi per tutti i nodi e infine per il clone di i, poi va in end.
  Per ottenere il ciclo in G, semplicemente passiamo direttamente per il nodo di partenza

senza attraversare il clone.

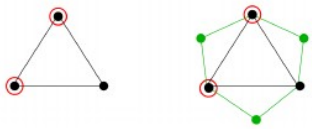
30A) Riduzione da VC (Ricoprimento) a DOM (Insieme dominante)

Un $V' \subseteq V$ è detto **insieme dominante** se tutti i nodi in V sono raggiungibili in un passo a partire da nodi in V'. Nel nostro caso, vogliamo trovare un insieme dominante di dimensione k.

Vogliamo dimostrare che DOM sia NP-completo.

- Prima di tutto, dobbiamo dimostrare che DOM sia in NP. Possiamo dimostrarlo facilmente se consideriamo è molto facile verificare la soluzione.
- A questo punto, dobbiamo dimostrare che DOM sia NP-arduo, quindi dimostriamo che VC $\leq_P$ DOM.

Dimostrazione:
Dobbiamo dimostrare che G ammette un ricoprimento di dimensione al massimo k se e solo se G' ammette un insieme dominante di dimensione al più k'.
Per farlo, costruiamo G' a partire da G, aggiungendo un nodo per ogni arco, e lo colleghiamo all'estremità dell'arco.

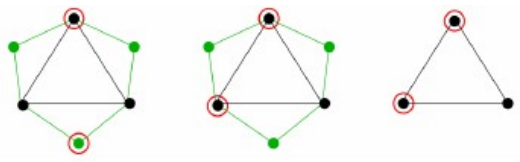


Il nuovo grafo G' avrà un numero di nodi pari agli archi + nodi di G, e un numero di archi pari a 3 volte quelli di G. Inoltre, k' sarà pari al valore di k'. La trasformazione di G in G' è **polinomiale**.

Procediamo con la dimostrazione:

- =>) Dobbiamo dimostrare che se G ammette un ricoprimento di dimensione $\leq$ k allora G' ammette un insieme dominante di dimensione $\leq$ k'.
  Se in G avevamo un ricoprimento, per definizione almeno uno dei nodi adiacenti ad ogni arco faceva parte dell'insieme dei nodi del ricoprimento.
  I nodi di G sono dunque raggiungibili in un passo a partire da questi nodi, siccome un ricoprimento è anche un insieme dominante sullo stesso grafo.
  Inoltre, anche i nodi **aggiunti** in G' sono raggiungibili in un passo. Dunque, un ricoprimento in G è anche un insieme dominante in G'.

- <=) Dobbiamo dimostrare che se il nuovo grafo G' ammette un insieme dominante di dimensione $\leq$ k', allora G ammette un ricoprimento di dimensione $\leq$ k.
  Supponiamo che G' ammetta un insieme dominante. Questo insieme potrebbe contenere dei nodi che non sono in G, tuttavia possiamo avere un insieme dominante anche considerando unicamente i nodi di partenza.
  Infatti, se prendiamo un nodo del grafo di partenza G al posto di prendere uno dei nuovi nodi aggiunti in G', riusciamo comunque a raggiungere in un passo tutti i nodi del "triangolo" composto in G', e quindi tutti i nodi di G', tra cui anche quelli aggiunti in più.
  Siccome per ogni arco in G abbiamo aggiunto un nodo in più in G', e ogni nodo che dell'insieme dominante in G' sta anche in G, possiamo pensare che ciascuno degli archi di G corrisponda a un nodo aggiunto in G', e quindi se raggiungiamo il nodo in più in G', sicuramente copriamo anche l'arco ad esso associato in G.



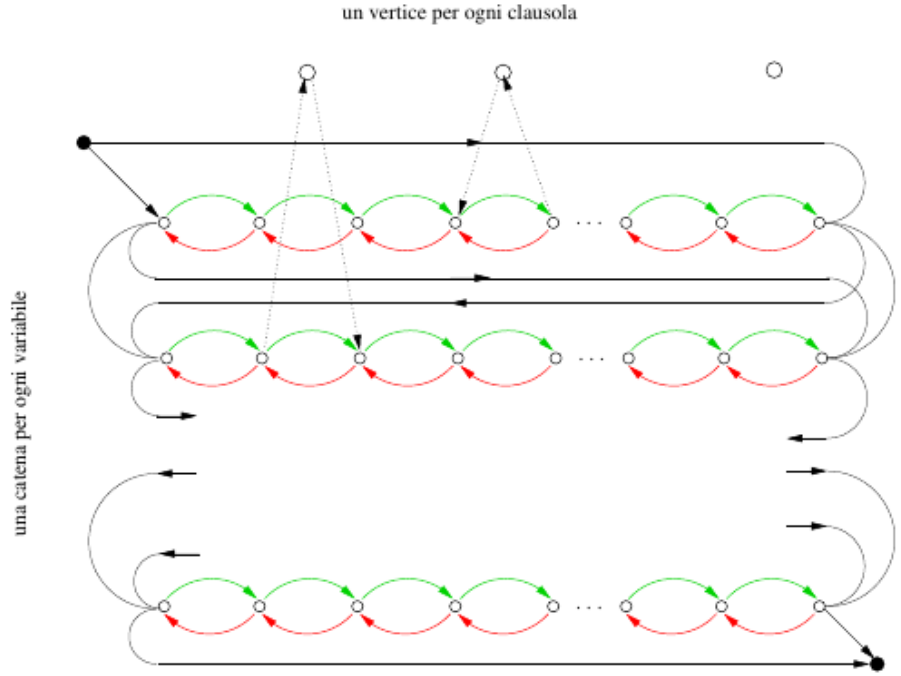
NOTARE BENE: *VC* COPRE GLI ARCHI, *DOM* I NODI.

31A) SAT ≤ Cammino hamiltoniano diretto

Un cammino hamiltoniano diretto è un cammino hamiltoniano su un grafo orientato. Vogliamo dimostrare che un formula SAT a clausole F=C1...Cm è soddisfacibile se e solo se il grafo G ammette un cammino hamiltoniano diretto.

Per permettere questa riduzione, costruiamo delle catene, **una per ogni variabile**, e di lunghezza 2m + 2 (dove *m* è il numero totale delle clausole di F).

A questo "sistema di catene", introduciamo un nodo di partenza e uno di fine, collegati con la prima e l'ultima catena rispettivamente.
Consideriamo poi che siccome possiamo attraversare queste catene sia da destra che da sinistra, dobbiamo collegare ogni estrermità della catena sia con il nodo più a destra che con ogni nodo più a sinistra di ogni catena precedente/successiva.



Aggiungiamo poi un vertice per ogni clausola. Possiamo vedere ogni clausola come una colonna della catena. I nodi corrispondenti alle clausole sono collegati in questo modo alle catene:

- Se nella clausola Ci la variabili X non è negata, allora creiamo due archi che collegano il nodo con due estremi della catena da sinistra verso destra.
- Altrimenti, se nella clausola Ci la variabile X è negata, i due archi vanno da destra verso sinistra.

In questo modo, se stiamo attraversando la catena in senso positivo (e quindi la variabile associata ad essa ha valore *true*) e troviamo un collegamento con una clausola, possiamo dire che questa clausola sia soddisfatta per quella variabile. Il discorso è lo stesso se attraversiamo la catena da destra verso sinistra, e quindi la variabile associata ha valore false.

Adesso che abbiamo costruito il grafo, vogliamo dimostrare che F è soddisfacibile se e solo se il grafo ammette un cammino hamiltoniano.

=>) Supponiamo che F sia soddisfacibile. Dimostrare che il grafo ammette un cammino hamiltoniano è banale, siccome ce lo assicura la costruzione di G (infatti dobbiamo scorrere tutte le catene e salire verso i nodi associati alle clausole).

<=) Supponiamo ora che il grafo G ammetta un cammino hamiltoniano diretto.
Consideriamo ora li situazione in cui questo cammino abbia una topologia tale per cui, partendo da un nodo ***u*** della catena e risalendo un nodo ***c*** della clausola, possiamo spostarci in un'altra catena.
Questo cammino però non ammette un cammino hamiltoniano, infatti l'unico modo in cui possiamo attraversare il nodo ***u'*** della catena adicante ad *u*, sarebbe passando per un nodo adiacente (che sia u o c, o un altro nodo ***u''***). Se però attraversassimo ***u'*** da ***u''***, il cammino che si otterrebbe non ammetterebbe un cammino hamiltoniano: visto che i nodi u e da quello da cui si è arrivati sono già stati attraversati, allora il cammino semplice non può proseguire.
L'unico modo di attraversare è u' è quindi riscendendo direttamente da c, e il cammino, allora, deve necessariamente seguire ciascuna catena l'una dopo l'altra.
Se il cammino esiste, allora avere questa formula ci assicura che tutte le clausole sono raggiunte, e per come è costruito G significa che sono tutte soddisfatte.

32A) VC ≤ Hitting Set

Dato un insieme S, e dato un insieme di insiemi C = $S_1..S_n$, dove ∀ $S_i \subseteq S$, diciamo che H è un hitting set per C se

$$\forall i \,.\, H \cap Si \neq \varnothing$$

Generalmente, noi siamo interessati a cercare l'hitting set minore.

Dimostriamo ora che il problema di determinare l'esistenza di un hitting set di cardinalità ≤ k sia NP-completo.
Prima di tutto possiamo notare come il problema HS sia in NP, siccome possiamo facilmente verificare la sua esistenza usando H come certificato, e lo spazio di ricerca è $2^{|S|}$.

Dimostriamo che è NP-arduo usando VC.

Dobbiamo considerare ogni sott'insieme S_i come un **iperarco** (ovvero un arco con un numero arbitrario di endpoint).
Per ogni iperarco, vogliamo prendere almeno un nodo che vi è connesso ad esso, quindi se abbiamo un ricoprimento, allora ogni sott'insieme è colpito.

La riduzione viene fatta in questo modo:

- Sia G = (V, E)
- Costruiamo l'insieme *C* formato da coppie {u, v} per ogni arco in E.
- Per definizione, un ricoprimento copre tutti gli archi, quindi ogni coppia è coperta e dunque è anche un hitting set per C, con k che non cambia tra VC e HS.

33A) HAM-C ≤ Commesso viaggiatore
Il problema del commesso viaggiatore TSP consiste nel determinare l'esistenza o meno di un cammino di lunghezza al più k in un grafo completamente connesso in n citta, dove la distanza fra città e città è $d_{i,j}$, e che permette di toccare tutte le città

Questo problema è in NP siccome è di facile verifica, avendo una soluzione come certificato.

Per dimostrare che sia NP-arduo, facciamo una riduzione da HAM-C a Commesso viaggiatore.

Consideriamo gli input di ciascun problema, che sono:

- G per HAM-C
- G', k e una matrice delle distanze per TSP.

Costuiamo quindi G', che ha gli stessi nodi di G ma è completamente connesso. Ad ogni arco attribuiamo questi pesi:

$$d_{i,j} = \begin{cases} 1 & (i,j) \in G \\ a > 1 & (i,j) \notin G \end{cases}$$

L'idea è questa:

- Se esiste un arco G che collega due nodi, allora gli diamo peso 1 in G'.
- Altrimenti gli diamo un peso maggiore di 1.

In questo modo, se esiste un ciclo hamiltoniano in G, allora esiste un ciclo di lunghezza n in G', e se esiste un ciclo di lunghezza n in G', allora questo è un ciclo hamiltoniano in G (siccome deve aver toccato tutti i nodi usando solo gli archi che erano in G).

35A) Mezza cricca

Il problema della mezza cricca consiste nel trovare una cricca di dimesione n/2, dove è n la dimensione dei nodi del grafo.

Facciamo la riduzione dal problema della cricca a quello della mezza cricca. Ovvero, esiste in G una cricca di dimensione k se e solo se esiste in G' una cricca di dimensione n/2.
Consideriamo gli input dei due problemi. Questi sono

- (G, k) per il problema della cricca
- G' per il problema della mezza cricca.

Concettualmente, per fare questa riduzione, vogliamo aggiungere dei nodi a G per ottenere G', e questa trasformazione avviene in base a k. Supponiamo ora che G = (V, E), |V| = n.
Abbiamo due casi da considerare:

- **Caso k ≤ n/2**. Consideriamo per semplicità il caso in cui n = 5 e k = 2.
  L'idea è di aggiungere un nodo completamente connesso a tutti gli altri, in modo che questo vegna così a far parte della cricca.
  In questo modo se esiste una cricca di dimensione 2 in G, abbiamo in G' una cricca di dimensione 3; mentre se abbiamo in G' una cricca di dimensione 3, allora se questa continua ad essere completamente interna a G, in G continua ad esiste una cricca di dimensione *maggiore o uguale* a k=2, altrimenti se anche il nodo aggiunto vi ci faceva parte, in G la dimensione è 2.
  Se non abbiamo dei n e k fissati (come nell'esempio appena fatto), sarà necessario capire il numero di nodi da aggiungere. Consideriamo che G'= (V', E'), dove |V'| = n+p, con *p* che sarebbe il numero di nodi da aggiungere. Quindi, la cricca in G' ha dimensione **p + k** .
  Siccome vogliamo che questa quantità sia esattamente la metà di |V'|, abbiamo che:
  $$\mathbf{2(k + p) = n + p \rightarrow p = n - 2k}$$
  La quantità è positiva siccome k ≤ n/2

- **Caso k ≥ n/2**. In questa la dimensione della cricca è troppo grande. Quindi, al posto di estendere la cricca, **aggiungiamo dei nodi isolati**. In particolare, p nodi isolati in modo che **p = 2k – n** (siccome n + p = 2k). Questa quantità non è negativa siccome k ≥ n/2.

35A) Knapsack
su appunti

36A) $NP^{NP} = NP \iff NP = coNP$
su appunti

37A) Gerarchia polinomiale: spiegare cos’è
su appunti