Informatica Teorica 2022/2023 - Esercitazione 3

29 Marzo 2023

Sessione I

Nozioni Richieste. Complessitá di Tempo; Notazione Asintotica.
Sia M una TM che ferma su ogni input. La sua *complessitá di tempo* é definita come $f : \mathbb{N} \to \mathbb{N}$, dove $f(n)$ é il massimo numero di passi che M impiega a fermarsi su input arbitrario di lunghezza n.
Date le funzioni $f, g : \mathbb{N} \to \mathbb{N}$ scriviamo $f(n) = O(g(n))$ e diciamo che $g(n)$ é un *asymptotic upper bound* se esistono $c, m \in \mathbb{N}$ tali che, per ogni $n \geq m$, $f(n) \leq cg(n)$. In altre parole, $g(n)$ é sempre grande almeno quanto $f(n)$ per n sufficientemente grandi e modulo un fattore costante.

Problema 1. Dato il linguaggio:

$$A = \{\texttt{0}^k\texttt{1}^k \mid k \geq 0\}$$

considera la (multi-tape) TM M che decide A e programmata come segue:

1. M legge l'input e rigetta se trova uno `0` a destra di un `1`.
2. M legge gli `0`i sul nastro 1 e li copia sul nastro 2.
3. M legge gli `1`i sul nastro 1 e per ogni `1` sul nastro 1 cancella uno `0` sul nastro 2. Se tutti gli `0`i sono cancellati prima che tutti gli `1`i siano letti, M rigetta.
4. Se tutti gli `0`i sono cancellati, M accetta; se qualche `0` resta sul nastro 2, M rigetta.

(a.) Qual'é la *time complexity* di M? (b.) Ed espressa in notazione asintotica?

Soluzione 1. (a.) Notiamo che:[1]

1. Leggere l'input richiede n **passi**; (eventualmente) rigettare richiede **1 passo**.
2. Tornare indietro richiede n **passi**.
   Leggere gli `0`i sul nastro 1 e copiarli richiede n **passi**.
3. Tornare indietro richiede n **passi**.
   Leggere gli `1`i sul nastro 1 e per ogni `1` sul nastro 1 cancellare lo `0` sul nastro 2 richiede n **passi**; (eventualmente) rigettare richiede **1 passo**.

[1] Osserva che questa forma di analisi si basa su un astrazione e ha come obiettivo l'individuazione di un limite alla complessitá in tempo dell'algoritmo. In questo senso puó essere piú o meno "raffinata" (prestando sempre attenzione a considerare sempre il caso peggiore). Nota anche che l'analisi viene effettuata sulla procedura presentata, che puó non essere ottimale (come el caso in esame). In esercizi come questo la compessitá va considerata in relazione all'algoritmo dato, non al migliore possibile (per esempio, in questo caso 3. é definita in modo tale da rileggere *integralmente* l'input, dal quale non sono stati cancellati bit.

4. Accettare richiede **1 passo**; rigettare richiede **1 passo**.

La procedura non viene ripetuta quindi l'esecuzione richiede al piú $n+2n+2n+1 = 5n+1$ **passi**.
(b.) In notazione asintotica,

1. Leggere l'input; (eventualmente) rigettare richiede complessivamente $O(n)$ **passi**.
2. Tornare all'inizio, leggere gli `0`i sul nastro 1 e copiarli richiede complessivamente $O(n)$ **passi**.
3. Tornare all'inizio, leggere gli `1`i sul nastro 1 e per ogni `1` sul nastro 1 cancellare lo `0` sul nastro 2 richiede, (Eventualmente) rigettare richiede complessivamente $O(n)$ **passi**.
4. Accettare o rigettare richiede $O(1)$ **passi**.

La complessitá complessiva é dunque $O(n) + O(n) + O(n) + O(1) = O(n)$.

La *time complexity* richiesta per decidere A dipende anche dal modello scelto (in questo caso multi-tape TM). Cf. questo algoritmo con quello presentato in classe (Lezione 11) di complessitá $O(n^2)$.

Nozioni Richieste. Caratterizzazion*i* di **NP**.
Data una funzione $f : \mathbb{N} \to \mathbb{N}$ definiamo la classe di complessitá (di tempo) NTIME($t(n)$) come la collezione di tutti i linguaggi decidibili da NTM (a un nastro) in tempo $O(t(n))$:

$$\mathbf{NP} = \bigcup_k NTIME(n^k).$$

(Caratterizzazione Alternativa) Un linguaggio L é *verificabile* se esiste una TM M (che termina sempre, accettando o rigettando) tale che:

$$w \in L \quad sse \quad esiste\ w'\ t.c.\ M\ accetta\ \langle w, w' \rangle.$$

Inoltre se M esegue in tempo polinomiale diciamo che L é verificabile polinomialmente. Intuitivamente c'é un *certificato* del fatto che w sia in L.

Problema 2. Considera il problema che dato un insieme di numeri $x_1, \ldots, x_k$ e un target t, determina se l'insieme dato contiene un sottoinsieme la cui somma di elementi ha valore t:

$$SSUM = \{\langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}\ \&\ per\ qualche\ \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \sum y_i = t\}.$$

Esempio. $\langle \{4, 11, 12, 21, 28, 50\}, 25 \rangle \in$ SSUM, in quanto 4 + 21=25.
Dimostra SSUM $\in$ **NP** (a.) tramite (poly-time) NTM e (b.) tramite verificatore.

Soluzione 2. (a.) Definiamo una poly-time NTM (ovvero un algoritmo non-deterministico che computa in tempo polinomiale) per SSUM come segue. Su input $\langle S, t \rangle$:

1. Seleziona non-deterministicamente un sottoinsieme S' di numeri di S.
2. Controlla se S' sia un insieme di numeri la cui somma ha valore t.
3. Se sì, accetta; altrimenti, rigetta.

(Ricorda che una NTM accetta l'input se almeno uno dei suoi percorsi di computazione é accettate e la *time complexity* é data dal ramo di computazione piú lungo.)

(b.) Definiamo un verificatore *polinomiale* per SSUM. L'idea é che il certificato C sia il sottoinsieme di S i cui elementi sommati abbiano valore t. Definiamo V che su input $\langle\langle S, t\rangle, C\rangle$:

1. Controlla che S contenga tutti i numeri in C.
2. Controlla che C sia un insieme di numeri che sommati diano t.
3. Se entrambi i test hanno esito positivo, accetta; altrimenti rifiuta.

L'esecuzione di V richiede tempo polinomiale.
(Ricorda che un *verificatore* per un linguaggio L é un algoritmo V dove $L = \{w \mid V \text{ accetta } \langle w, c\rangle$, per qualche stringa c$\}$. Misuriamo il tempo del verificatore solo in termini della lunghezza di w, quindi un *verificatore poly-time* esegue in tempo polinomiale nella lunghezza di ogni possibile input w.)
Come abbiamo visto si puó convertire il verificatore poly-time in un equivalente poly-time NTM e viceversa: la NTM simula il verificatore indovinando il certificato; il verificatore simula la NTM considerando il ramo accettante come certificato.

Nozioni Richieste: Poly-Riduzione; **NP**-Completezza.
Siano L e L' linguaggi su alfabeto Σ. Diciamo che L é *poly (mapping-)reducible* a L', $L \leq_p L'$, se esiste una TM che computa in tempo polinomiale una funzione (totale) $f : \Sigma^* \to \Sigma^*$ tale che:

$$x \in L \quad sse \quad f(x) \in L'.$$

Detto altrimenti, $L \leq_p L'$ se $L \leq L'$ e la riduzione corrispondente é computabile in tempo polinomiale.
Un linguaggio L é **NP**-completo se é in **NP** e ogni altro linguaggio $L' \in$ **NP** é poly-riducibile ad esso.

Problema 3. Se L é **NP**-completo, $L' \in$ **NP** e $L \leq_p L'$, allora L' é **NP**-completo.
Suggerimento. Ricorda che durante la passata esercitazione abbiamo dimostrato che la mapping-reduction é transitiva.

Soluzione 3. Per Df. di **NP**-completezza, per dimostrare che L' é **NP**-completo dobbiamo mostrare: (i) $L' \in$ **NP** e (ii) per ogni $L^* \in$ **NP**, $L^* \leq_p L'$. La condizione (i) vale per ipotesi. Dunque dobbiamo dimostrare solamente (ii).
Consideriamo un linguaggio generico $L'' \in$ **NP**. Per ipotesi, sappiamo che L é **NP**-completo, dunque, in particolare, $L'' \leq_p L$ (A1). Per ipotesi sappiamo anche che $L \leq_p L'$ (A2).
Inoltre la poly-riduzione é u caso particolare di mapping-reduction, la cui riduzione é poly-time. Applicando lievi modifiche alla dimostrazione della trasitivitá per la mapping-riduzione (cf. Esercitazione 15 Marzo), dimostriamo che anche $\leq_p$ é transitiva (la riduzione costruita da funzioni di riduzioni poly-time per ipotesi, é ancora poly-time).[2] In particolare dati $L'' \leq_p L$ (A1) e $L \leq_p L'$ (A1), possiamo concludere $L'' \leq_p L'$.

[2]Infatti, la TM che computa h simula la prima poly-time TM (ipotesi) ottenendo un valore y; poi, simula una seconda poly-time TM (ancora per ipotesi) su tale y. Dunque la TM che esegue h, definendo la riduzione desiderata, esegue ancora in tempo polinomale.

Poiché L'' é un linguaggio generico in **NP** e ($L' \in$ **NP** per ipotesi), concludiamo anche che L' é **NP**-completo.

Nozioni Richieste: Formule Booleane; Formula Soddisfacibile; Letterale; Clausole; CNF; SAT; 3SAT; Teorema di Cook-Levin.
Una *formula Booleana* é costituita da variabili $x, y, z, \ldots$, le loro negazioni $\neg x, \neg y, \neg z, \ldots$, congiunzioni e disgiunzioni di letterali. Tali variabili hanno valore 1 o 0. Una formula Booleana é *soddisfacibile* se esiste un assegnamento di valore alle sue variabili che le dia valore 1. Una formula Booleana é una *clausola* se é una disgiunzione di letterali. Una formula Booleana é in forma normale congiuntiva (CNF) se é una congiunzione di clausole. Una formula Booleana é in 3CNF se é in CNF e ogni clausola contiene esattamente tre letterali.

$$SAT = \{\langle F \rangle \mid F \; formula \; Booleana \; sodddisfacibile\}$$

$$3SAT = \{\langle F \rangle \mid F \; formula \; Booleana \; 3CNF \; soddisfacibie\}$$

Teorema di Cook-Levin (1971): SAT é **NP**-completo.

Problema 4.1. Sapendo che 3SAT $\in$ **NP**, mostra (ad alto livello) che 3SAT é **NP**-completo (utilizzando il teorema di Cook-Levin).
Suggerimento. Considera il problema precedente, insieme alla definizione di **NP**-completezza e il teorema di Cook-Levin.

Soluzione 4.1. Per dimostrazione precedente se L é **NP**-completo e $L' \in$ **NP** é tale che $L \leq_p L'$, allora L' é **NP**-completo (Problema 3).
Per ipotesi, sappiamo giá che 3SAT $\in$ **NP**.
Inoltre, per il teorema di Cook-Levin sappiamo che SAT é **NP**-completo.
Dunque, per dimostrare che 3SAT é **NP**-completo sarebbe sufficiente dimostrare che SAT $\leq_p$ 3SAT, ovvero, per Df. di poly-riduzione, che esiste una *poly-time* funzione computabile (totale) f tale che

$$x \in SAT \quad sse \quad f(x) \in 3SAT.$$

Nozioni Richieste: Space Complexity; PSPACE; TQBF.
Sia M una TM che ferma su ogni input. La *space complexity* di M é la funzione $t : \mathbb{N} \to \mathbb{N}$, tale che $t(n)$ é il numero massimo di celle che M scansiona per ciascun input di lunghezza n.
Data una funzione $t : \mathbb{N} \to \mathbb{N}$ definiamo la classe di complessitá di spazio SPACE($t(n)$) come la collezione di tutti i linguaggi da una TM (deterministica, a un nastro) in tempo $O(t(n))$:

$$SPACE(t(n)) = \{L \mid L \; deciso \; da \; TM \; in \; O(t(n))\text{-}space\}.$$

PSPACE é la classe di linguaggi decidibili da poly-space TM:

$$PSPACE = \bigcup_k SPACE(n^k).$$

É stato dimostrato in classe (Lezione 13) che TQBF é PSPACE-completo.

Problema 5.1[3] In generale, un *game* é una competizione tra giocatori opponenti che tentano di raggiungere un obiettivo seguendo date regole. Consideriamo il *formula game* (fg). Sia F una QBF (quantified Boolean formula) in PNF (prenex normal form):

$$F = \exists \mathbf{X_1} \forall \mathbf{X_2} \exists \mathbf{X_3} \ldots Q \mathbf{X_k} G$$

con $Q \in \{\forall, \exists\}$. I due giocatori – diciamo `A` ed `E` – selezionano *a turno* i valori di veritá da attribuire a (gruppi) di variabili. In particolare, `A` assegna i valori alle variabili vincolate da $\forall$ ed `E` a quelle vincolate da $\exists$. A partire dai valori dati si decide chi vince: se $G = 1$, vince `E`; se $G = 0$, vince `A`.

(a.) Consideriamo un semplice esempio Data la seguente formula

$$F_1 = \exists x_1 \forall x_2 \exists x_3 ((x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3))$$

Se `E` assegna $x_1 = 1$; poi `A` assegna $x_2 = 0$; `E` assegna $x_3 = 1$, chi vince?

Diciamo che un giocatore ha una *winning strategy* se, seguendo tale strategia, vince per ogni scelta dell'altro giocatore. (b.) Consideriamo

$$F_2 = \exists x \forall y (x \vee \neg y).$$

Il giocatore `E` ha una *winning strategy* su F_2? Se sì, definiscila.
Considerando anche

$$F_3 = \exists x_1 \forall x_2 \exists x_3 ((x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)).$$

Il giocatore `A` ha una *winning strategy* per F_3? Se sí, definiscila.

Soluzione 5.1. (a.) Vince `E`. Infatti, $[\![(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)]\!]_{\texttt{play}} = 1$, dove $[\![\cdot]\!]_{\texttt{play}}$ indica l'assegnamento definito sopra.
(b.) Sí `E` ha una *winning strategy* su F_2. Essa consiste semplicemente nell'assegnare valore 1 per x. Sí `A` ha una *winning strategy* su F_3. Infatti se `A` sceglie $x_2 = 0$ falsifica la matrice, indipendentemente dalla scelta di `E`

Problema 5.2 Consideriamo il problema di determinare quale giocatore abbia una *winning strategy* in un *formula game* associato a una data QBF F:

$$FG = \{\langle F \rangle \mid \texttt{E}\ ha\ winning\ strategy\ nel\ fg\ associato\ a\ F\}$$

Mostra (anche informalmente) che FG é PSPACE-completo.

Soluzione 5.2 Una formula é vera esattamente quando `E` ha una *winning strategy* nel *formula game* associato. Infatti una QBF della forma

$$\exists \mathbf{X_1} \forall \mathbf{X_2} \exists \mathbf{X_3} \ldots G$$

é vera quando esiste qualche assegnamento per $\mathbf{X_1}$ tale che per ogni assegnamento di $\mathbf{X_2}$, esiste un assegnamento per $\mathbf{X_3}$ ... tale per cui G sia vera per tali assegnamenti. Analogamente, `E` ha una

[3] Le definizioni fornite sono semplificazioni, funzionali a introdurre questo esercizio e dare un'idea intuitiva del rapporto tra logiche proposizionali quantificate e (modellizzazioni) dei giochi. La letteratura in merito é vasta (un esempio di articolo "classico" (e breve) sul tema é: C.H. Papadimitriou, *Game Against Nature*, *Journal of Computer and System Sciences*, 31/2, pp. 288-301, 1985.

strategia vincente, ovvero assegna valori a $\mathbf{X}_1$ tali che per ogni scelta messa in atto da `A` per di $\mathbf{X}_2$ ... il valore di G sia 1. Quindi per ogni QBF F, $F \in$ TQBF esattamente quando $F \in$ FG.
Detto alltrimenti, FG altro non é che una diversa formulazione di TQBF. Infatti per ogni QBF F:

$$F = 1 \quad sse \quad \texttt{E}\ ha\ winning\ strategy\ su\ F.$$

É stato dimostrato in classe che TQBF é PSPACE-completo (Lezione 13). Quindi, chiaramente, anche FG é PSPACE-completo.

Sessione II

Problema 6. Dato il linguaggio

$$U = \{w \in \Sigma^* \mid w\ contiene\ eguale\ numero\ di\ \texttt{0}\ e\ \texttt{1}\},$$

considera la TM M su alfabeto $\Sigma = \{\texttt{0}, \texttt{1}\}$ in grado di "contrassegnare" i simboli dell'alfabeto e tale che, su input w:

1. Scansiona il nastro fino al primo bit (non contrassegnato) `0` o `1`

   Se non ne trova alcuno, accetta

   Altrimenti continua a scansionare fino all primo bit diverso (`1` e `0` resp.)
2. Se non ne trova, rigetta; Altrimenti segna i due simboli e ripete la procedura

(Per la finalitá dell'esercizio lavoriamo con astrazioni di TM; l'estensione dell'alfabeto della macchina a "bit contrassegnato" non inficia il calcolo della *time complexity* riferita alla M qui descritta.) (a) Qual'é la complessitá in tempo di M in notazione asintotica?

Soluzione 6.1 (Nota che stiamo considerando un'astrazione della TM, e ció che interessa é la definizione di un *upper bound* del tempo di esecuzione.) Osserva preliminarmente che, per M

1. Scansionare il nastro fino a `0` o `1` richiede **al massimo n passi**.

   (eventuamente) accettare richiede **al massimo 1 passo**.

   Continuare a scansionare fino al bit corrispondente richiede **al massimo n passi**.
2. (eventualmente) rigettare richiede **al massimo 1 passo**;

   (contrassegnare i simboli e) tornare all'inizio per ripetere la procedura richiede **al massimo n passi**.

La procedura complessiva é **ripetuta al piú $\frac{n}{2}$ volte**. Quindi avremo al massimo $2n + 2$ passi per al piú $\frac{n}{2}$ volte. Concludiamo che l'esecuzione complessiva richiede al piú $n^2 + n$ passi.
In notazione asintotica,

1. Scansionare il nastro fino a `0` o `1` richiede al piú $O(n)$ passi.
2. Accettare richiede al piú $O(1)$ passi.
3. Continuare la scansione richiede al piú $O(n)$ passi.
4. Rigettare richiede al piú $O(1)$ passi.
5. Tornare all'inizio richiede al piú $O(n)$ passi.

La procedura puó essere ripetuta al piú $O(n)$ volte. Concludiamo che essa globalmente richiede al piú $O(n)O(n) = O(n^2)$ passi.

Problema 6.2. Dato il linguaggio

$$U = \{w \in \Sigma^* \mid w \textit{ ha uguali } \texttt{0} \textit{ e } \texttt{1}\},$$

considera la multi-tape TM M' su alfabeto $\Sigma = \{\texttt{0}, \texttt{1}\}$, tale che su input w:

1. Scansiona il nastro e copia tutti gli `1`i presenti sul nastro secondario
2. Muove entrambe le testine all'inizio dei rispettivi nastri
3. Fino al raggiungimento della fine dell'input:
   - Scansiona il nastro in input fino a `0`
   - Associa `0` a `1` del nastro secondario
4. Se non si ha corrispondenza uno-a-uno, rigetta; Se ciascuno `0` é associato al corrispondente `1`, accetta.

Qual'é la complessitá in tempo di M espressa in notazione asintotica?

Soluzione 6.2. L'esecuzione totale richiede $O(n)$ passi. Infatti, considera che:

1. Scansionare il nastro e copiare tutti gli `1`i su nastro secondario richiede **O(n) passi**.
2. Muovere entrambe le testine all'inizio dei rispettivi nastri richiede **O(n) passi**.
3. Fino al raggiungimento della fine dell'input scansionare il nastro fino a `0` e associare con `1` del nastro secondario, richiede **O(n) passi**.
4. Rigettare o accettare, in base alla corrispondenza richiede **O(1) passi**.

Problema 7. Dimostra PATH $\in$ **P** (senza consultare le slide).
Un algoritmo brute-force per PATH potrebbe fare al caso nostro? Motiva.
Suggerimento. Ricorda che, dato un grafo diretto G con nodi s e t, il problema PATH consiste nel determinare se esiste un percorso diretto da s a t:

$$PATH = \{\langle G, s, t\rangle \mid G \textit{ grafo diretto con percorso diretto da } s \textit{ a } t\}.$$

Soluzione 7. Presentiamo un algoritmo poly-time che decide PATH (ed eviti la forza bruta). Contrassegnamo successivamente tutti i nodi in G ottenibili da s con percorso diretto.
Definiamo un algoritmo poly-time M per PATH come segue. Sia G un grafo diretto di nodi s e t:

1. Contrassegna il nodo s.
2. Ripete la seguente procedura finché nessun nuovo nodo é contrassegnato;
   Scansiona tutti gli archi di G: se trova un arco (a, b) da nodo contrassegnato a a nodo non contrassegnato b contrassegna b.
3. Se t é contrassegnato accetta; altrimenti rifiuta.

Analizziamo la *time complexity* di questo algoritmo per dimostrare che é polinomiale. 1. e 3. sono eseguiti una sola volta. Sono entrambi implementati in tempo polinomiale su modello deterministico ragionevole. 2. é eseguito al massimo m volte, dove m é il numero dei nodi di G. Esso richiede una scansione dell'input e il controllo se certi nodi siano contrassegnati, il che é ancora implementabile in tempo polinomiale: $O(1) + O(n * n^2) + O(1) = O(n^3)$. Quindi M descrive un algoritmo polinomiale per PATH

L'algoritmo brute-force procederebbe esaminando tutti i sentieri potenziali in G e determinando se qualcuno di questi sia un percorso diretto da s a t. Un percorso potenziale é una sequenza di nodi in G con lunghezza al massimo m, dove m é il numero dei nodi in G. Se esiste un percorso diretto da s a t, ha lunghezza al piú m. Ma il numero di tali percorsi é approssimativamente m^m, che é esponenziale nel numero dei nodi di G. Quindi questo algoritmo brute-force richiede tempo esponenziale.

Problema 4.2. Sia $F = \bigwedge_{j\in\{1,\ldots,n\}} l_j$

$$F \in SAT \quad sse \quad f(F) \in 3SAT.$$

Suggerimento. Considera la funzione ausiliaria g tale che, per ogni letterale l, $g(l) = (l \vee x_1 \vee x_2) \wedge (l \vee x_1 \vee \overline{x_2}) \wedge (l \vee \overline{x_1} \vee x_2) \wedge (l \vee \overline{x_1} \vee \overline{x_2})$.

Soluzione 4.2. Dimostriamo che, per ogni l, l é logicamente equivalente a $g(l)$, ovvero, per ogni valutazione v:

$$v(l) = 1 \quad sse \quad v(g(l)) = 1.$$

Questa dimostrazione puó essere ottenuta in vari modi. Possiamo, per esempio, considerare la tavola di veritá di l e $g(l)$: dove $C1 = (l \vee x_1 \vee x_2)$, $C2 = (l \vee x_1 \vee \overline{x_2})$, $C3 = (l \vee \overline{x_1} \vee x_2)$ e $C_4 = (l \vee \overline{x_1} \vee \overline{x_2})$.

Consideriamo ora una funzione f, tale che, per ogni $F = \bigwedge_{j\in\{1,\ldots,n\}\}} l_j$, $f(F) = \bigwedge_{j\in\{1,\ldots,n\}} g(l_j)$. (Chiaramente, la sua esecuzione richiede tempo polinomiale.)
Inoltre, le congiunzioni di formule *equivalenti* sono a loro volta equivalenti, per cui,

$$v(F) = 1 \quad sse \quad v(g(F)) = 1.$$

Chiaramente, F é soddisfacibile se e solo se $f(F)$ é soddisfacibile. Inoltre, per costruzione $f(F)$ é in 3CNF. Concludiamo allora che per ogni $F = \bigwedge_{j\in\{1,\ldots,n\}} l_j$:

$$F \in SAT \quad sse \quad f(F) \in 3SAT.$$

Problema 8. (Senza consultare le slide) Dimostra che **NP** $\subseteq$ PSPACE.
Suggerimento. Dimostra che (1) SAT $\in$ PSPACE e (2) per ogni A, se $A \in$ **NP** allora $A \in$ PSPACE.
Soluzione 8. SAT $\in$ PSPACE. Costruiamo una TM M per decidere SAT. Intuitivamente possiamo risolvere SAT provando ogni possibile assegnamento di valore di veritá alle variabili che occorrono nella formula. Questo richiede molto tempo ma non molto spazio.
Sia F una formula Booleana, definiamo TM M tale che su input $\langle F \rangle F$:

1. Per ogni assegnamento di valore di veritá delle variabili $x_1, \ldots, x_m$, valuta F su tale assegnamento; se F ha valore 1, M accetta;
2. Altrimenti M rigetta.

Osserva che la macchina riutilizza lo spazio per i nuovi assegnamenti di veritá. Infatti, una volta che si é controllato l'assegnamento, questo puó essere cancellato dal nastro e lo spazio puó essere riutilizzato per il controllo del'assegnamento successivo.
M esegue in spazio lineare. Infatti ciascuna iterazione del loop riutilizza la stessa porzione dello spazio: l'assegnamento dei valori di veritá richiede spazio $O(m)$ (dove m é il numero delle variabili); il numero delle variabili é al piú $O(n)$ dove n é la lunghezza dell'input. Dunque la macchina esegue in spazio $O(n)$. Nota che invece la time complexity é molto grande (esponenziale).

Mostriamo ora che per ogni $L \in$ NP, $L \in$ PSPACE. Osserviamo che se $L \leq_p L'$ e $L' \in$ PSPACE, allora $L \in$ PSPACE. Ma SAT é **NP**-completo (teorema Cook-Levin), quindi per ogni $L \in$ **NP**, $L \leq_p$ SAT. Quindi $L \in$ PSPACE.

Possiamo dunque concludere che **NP** $\subseteq$ PSPACE.