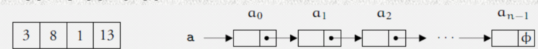
Problemi computazionali

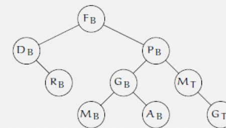
Intrattabilità, indecidibilità, e classi computazionali

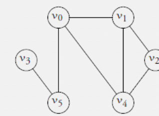
Algoritmo

- Essenza computazionale di un programma che ne descrive i passi fondamentali (più implementazioni)
- Ingredienti: **array, liste, alberi e grafi**
- Servono a strutturare i dati elementari: **bit, caratteri, interi, reali e stringhe**.
- Rappresentano istanze di problemi computazionali reali e concreti

3	8	1	13







Problemi indecidibili

- Non tutti i problemi computazionali ammettono algoritmi di risoluzione: **problema della fermata** (Halting problem, Turing, 1937)
- Terminologia moderna: dato un generico algoritmo (o programma) A in ingresso, esso **termina** o **va in ciclo**?

Termina?

```
1  Primo( numero ):
2    fattore = 2;
3    WHILE (numero % fattore != 0 )
4      fattore = fattore + 1;
5    RETURN (fattore == numero);
```

- Sì, perché la variabile `fattore` è incrementata di 1 e a un certo punto deve verificare la guardia
- [alvie]: http://www.algoritmica.org

Termina?

```
1  CongetturaGoldbach( ):
2    n = 2;
3    DO {
4      n = n + 2;
5      controesempio = TRUE;
6      FOR (p = 2; p <= n-2; p = p + 1) {
7        q = n - p;
8        IF (Primo(p) && Primo(q)) controesempio = FALSE
9      }
10   } WHILE (!controesempio);
11   RETURN n;
```

- Cerca il più piccolo n≥ 4 pari che non sia la somma di due primi
- Termina se e solo se trova n ≥ 4 per cui non esistono due primi p e q t.c. n = p + q
- Termina se e solo se confuta la congettura di Goldbach (problema aperto)

Problema della fermata

Non esiste un algoritmo per stabilire la terminazione di un **generico algoritmo / programma A**

1. Una sequenza di simboli può essere interpretata come **dato** o **programma**
2. Un programma può essere dato in pasto a un altro programma (es. compilatore)

Problema della fermata

3. Supponiamo che esista un algoritmo **Termina(A, D)** che, in tempo finito, restituisce
   Sì: A termina con input D
   No: A va in ciclo con input D

4. Osservazioni 1+2 implicano che è legale invocare Termina(A, A)

Problema della fermata

5. Consideriamo il seguente algoritmo

```
Paradosso(A)
    while( Termina(A, A) = Sì )
    ;
```

Domandone: **Paradosso(Paradosso)** termina?

CONTRADDIZIONE

Indecidibilità

- Il problema della fermata è quindi **indecidibile**, ossia non esiste alcun algoritmo di risoluzione.
- Altri problemi lo sono
  - Stabilire l'**equivalenza** tra due programmi (se per ogni possibile input, producono lo stesso output).

Decidibilità e Trattabilità

Problemi decidibili possono richiedere tempi di risoluzione elevati: **Torri di Hanoi**

- 3 pioli
- n = 64 dischi sul primo piolo (vuoti gli altri due), tutti di dimensione diversa
- Disco grande non può stare su piccolo
- Ogni mossa sposta un disco
- Obiettivo: spostarli tutti dal primo all'ultimo piolo

Soluzione ricorsiva

```
1 TorriHanoi( n, primo, secondo, terzo ):
2   IF (n = 1) {
3     PRINT primo ⟼ terzo;
4   } ELSE {
5     TorriHanoi( n - 1, primo, terzo, secondo );
6     PRINT primo ⟼ terzo;
7     TorriHanoi( n - 1, secondo, primo, terzo );
8   }
```

- [alvie]

Numero di mosse: 2^n-1

```
1 TorriHanoi( n, primo, secondo, terzo ):
2   IF (n = 1) {
3     PRINT primo ⟼ terzo;
4   } ELSE {
5     TorriHanoi( n - 1, primo, terzo, secondo );
6     PRINT primo ⟼ terzo;
7     TorriHanoi( n - 1, secondo, primo, terzo );
8   }
```

- Caso base:n = 1 --> $2^1-1 = 1$
- Passo induttivo: $(2^{n-1}-1)+1+(2^{n-1}-1) = 2^n-1$
- n=64 --> $2^{64}-1$ = 18 446 744 073 709 551 615
- 1 mossa/sec --> circa 585 miliardi di anni!
- NOTA: Tempo esponenziale, ed è stato dimostrato che non si può risolvere in meno mosse

Tempo esponenziale 2^n-1
(1 operazione/sec)

n	5	10	15	20	25	30	35	40	45
tempo	31s	17 m	9 h	12 g	1 a	34 a	1089 a	34865 a	1115689 a

Aumentare di un fattore **moltiplicativo** X (ossia X operazioni/sec) migliora **solo** di un fattore **additivo** $\log_2 X$

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	67	70	73	77	80	83	93

Nell'esempio: dischi gestibili nel tempo necessario a gestire 64 dischi

Tempo polinomiale

Torri di Hanoi generalizzate con k > 3 pioli

- Pioli numerati da 0 a k-1
- Obiettivo: spostare i pioli dallo 0 al k-1
- Hp. semplificativa: n è multiplo di k-2

```
1  TorriHanoiGen( n, k ):
2    FOR (i = 1; i <= k-2; i = i+1)
3      TorriHanoi(n/(k-2), 0, k-1, i);
4    FOR (i = k-2; i >= 1; i = i-1)
5      TorriHanoi(n/(k-2), i, 0, k-1);
```

Torri di Hanoi generalizzate

- Il codice richiede 2 x (k-2) x ($2^{n/(k-2)}$-1) mosse
- Al più n^2 mosse, fissando k = trunc(n/log n) e n ≥ 5
- n = 64 --> al più 64^2 = 4096 mosse

```
1  TorriHanoiGen( n, k ):
2    FOR (i = 1; i <= k-2; i = i+1)
3      TorriHanoi(n/(k-2), 0, k-1, i);
4    FOR (i = k-2; i >= 1; i = i-1)
5      TorriHanoi(n/(k-2), i, 0, k-1);
```

Tempo polinomiale n^2
(1 operazione/sec)

n	5	10	15	20	25	30	35	40	45
tempo	25 s	100 s	225 s	7 m	11 m	15 m	21 m	27 m	34 m

Aumentare di un fattore **moltiplicativo** X (ossia X operazioni/sec) migliora di un fattore **moltiplicativo** $\sqrt{X}$

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	202	640	2023	6400	20238	64000	2023857

Rappresentazione e dimensione dei dati

- Il bit segnala la presenza o l'assenza di un segnale
  - Rappresenta l'elemento costituente dell'informazione (informazione minima, due soli valori, 0 e 1)
  - Una stessa sequenza di bit può essere interpretata in molti modi (codice, dati, ecc.)
- **Numero di bit**: k bit codificano interi in $\{0 \ldots 2^k-1\}$

$$b_{k-1}\, b_1 \cdots b_0 \dashrightarrow n = \sum_{i=0,\ldots,k-1} b_i \times 2^i$$

Es. K = 3 `000,001,010,011,100,101,110,111`
`0 1 2 3 4 5 6 7`

Caratteri: 8 bit (ASCII) o 16 bit (Unicode/UTF8)

Reali: 32, 64 o 128 bit (segno,esponente, mantissa)

- Dimensione del dato: lunghezza della sua rappresentazione

Problemi

- Un problema è una relazione

$$P \sqsubseteq I \times S$$

I: insieme delle istanze in ingresso

S: insieme delle soluzioni

- Possiamo inoltre immaginare di avere un *predicato* che presa un'istanza $x \in I$ ed una soluzione $s \in S$, restituisce 1 se $(x,s) \in P$

Tipologie di problemi

- Problemi di decisione
  - Richiedono una risposta binaria
  - Un grafo è connesso? Un numero è primo?
  - Istanze positive ($(x,1) \in P$) o negative ($(x,0) \in P$)
- Problemi di ricerca
  - Richiedono di restituire una soluzione s tale che $(x,s) \in P$
  - Trovare un albero di copertura, trovare il mediano di un insieme di elementi

Tipologie di problemi

- Problemi di ottimizzazione
  - Data un'istanza x, si vuole trovare la migliore soluzione s tra tutte le possibili s per cui $(x,s) \in P$
  - Ricerca del minimo albero di copertura, ricerca del cammino minimo fra due nodi di un grafo

Problemi decisionali

- La teoria della complessità computazionale è definita principalmente in termini di problemi di decisione
  - Essendo la risposta binaria, non ci si deve preoccupare del tempo richiesto per restituire la soluzione e tutto il tempo è speso esclusivamente per il calcolo

Problemi decisionali

- Molti problemi di interesse pratico sono però di ottimizzazione
- È però possibile esprimere un problema di ottimizzazione in forma decisionale
  - Minimo albero di copertura: dato un grafo G ed un intero k, esiste un albero di copertura di G di costo al più k?
  - Problema non più difficile di quello di ottimizzazione
    - Se sappiamo trovare la soluzione ottima, la confrontiamo con k!

Problemi decisionali

- Il problema di ottimizzazione è quindi almeno tanto difficile quanto il corrispondente decisionale
  - Caratterizzare la complessità di quest'ultimo permette quindi di dare almeno una limitazione inferiore alla complessità del primo

Classi di complessità

- Dato un problema P ed un algoritmo A, diciamo che A risolve P se

A restituisce 1 su $x \Leftrightarrow (x,1) \in P$

- A risolve P in tempo $t(n)$ e spazio $s(n)$ se il tempo di esecuzione e l'occupazione di memoria di A sono rispettivamente $t(n)$ e $s(n)$

Classi Time e Space

- Data una qualunque funzione f(n), chiamiamo

Time(f(n)) e Space(f(n))

gli insiemi dei problemi decisionali che possono essere risolti rispettivamente in tempo e spazio O(f(n))

Classi Time e Space

- Il problema di verificare se un certo elemento è presente in un dizionario ordinato realizzato tramite array e contenente *n* elementi appartiene alla classi

Time(????) e Space(????)

Classi Time e Space

- Il problema di verificare se un certo elemento è presente in un dizionario ordinato realizzato tramite array e contenente *n* elementi appartiene alla classi

Time(log n) e Space(n)

- Estendiamo ora questa definizione….

Classe P

- *Algoritmo polinomiale (spazio o tempo): esistono due costanti c, $n_0 > 0$ t.c. il numero di passi elementari (celle di memoria utilizzate) è al più n^c per ogni input di dimensione n e per ogni $n > n_0$*

- La classe P è la classe dei problemi risolvibili in tempo polinomiale nella dimensione n dell'istanza di ingresso:

$$P=\cup_{c=0..\infty}Time(n^c)$$

Classe PSpace

- La classe PSpace è la classe dei problemi risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso:

  $PSpace=\cup_{c=0..\infty} Space(n^c)$

Classe ExpTime

- La classe ExpTime è la classe dei problemi risolvibili in tempo esponenziale nella dimensione n dell'istanza di ingresso:

  $ExpTime=\cup_{c=0..\infty} Time(2^{n^c})$

Relazioni

- Un algoritmo polinomiale può avere accesso al più a quante diverse locazioni di memoria (ordine di grandezza)?

Relazioni

- Un algoritmo polinomiale può avere accesso al più a quante diverse locazioni di memoria (ordine di grandezza)?
  - Polinomiale!
  - Quindi P $\sqsubseteq$ PSpace
- Inoltre risulta
  - PSpace $\sqsubseteq$ ExpTime
  - *Informalmente: assumendo che le locazioni di memoria siano binarie, n^c diverse locazioni di memoria possono trovarsi in al più 2^{n^c} stati diversi*

Relazioni

- Non è noto (ad oggi) se le inclusioni siano proprie
- L'unico risultato di separazione dimostrato finora riguarda P e ExpTime
  - Esiste un problema che può essere risolto in tempo esponenziale, ma per cui tempo polinomiale non è sufficiente

Esempi

- Tutti i problemi visti finora sono in P
- Il problema delle Torri di Hanoi in quale classe di complessità si trova?

Esempi

- Tutti i problemi visti finora sono in P
- Il problema delle Torri di Hanoi in quale classe di complessità si trova?
  - ExpTime
- Altro esempio interessante:
  - *Generazione di sequenze*

Genera le 2^n sequenze binarie

- Equivale a generare ricorsivamente tutti i sotto-insiemi di un insieme di n elementi (A[i]=1 sse l'i-esimo elemento e' selezionato)

Es. n = 3 `000,001,010,011,100,101,110,111`

`0   1   2   3   4   5   6   7`

- Struttura ricorsiva della generazione

```
000,100,010,110,001,101,011,111
000,100,010,110
000
```

fissa il bit a 1 e ricorri come per il bit a 0

Genera le 2^n sequenze binarie

- Equivale a generare ricorsivamente tutti i sotto-insiemi di un insieme di n elementi (A[i]=1 sse l'i-esimo elemento è selezionato)

```
1 GeneraBinarie( A, b ):
2   IF (b == 0) {
3     Elabora( A );
4   } ELSE {
5     A[b-1] = 0;
6     GeneraBinarie( A, b-1 );
7     A[b-1] = 1;
8     GeneraBinarie( A, b-1 );
9   }
```

invocato con b=n

[alvie]

Genera le n! permutazioni di A

a	b	c	d	a	b	d	c	a	d	c	b	d	b	c	a
b	a	c	d	b	a	d	c	d	a	c	b	b	d	c	a
a	c	b	d	a	d	b	c	a	c	d	b	d	c	b	a
c	a	b	d	d	a	b	c	c	a	d	b	c	d	b	a
c	b	a	d	d	b	a	c	c	d	a	b	c	b	d	a
b	c	a	d	b	d	a	c	d	c	a	b	b	c	d	a
			i = 3				i = 2				i = 1				i = 0

Per i = n-1, ..., 1, 0:

- scambia A[i] con quello in ultima posizione, A[n-1];
- i primi n-1 elementi di A sono ricorsivamente permutati **nella stessa maniera** (indipenden. da i);
- Scambia l'ultimo elemento A[n-1] con A[i] (per rimetterli a posto).

Genera le n! permutazioni di A

```
1  GeneraPermutazioni( A, p ):   ⟨pre: i primi p
2    IF (p == 0) {
3      Elabora( A );
4    } ELSE {
5      FOR (i = p-1; i >= 0; i = i-1) {
6        Scambia( i, p-1 );
7        GeneraPermutazioni( A, p-1 );
8        Scambia( i, p-1 );
9      }
10   }
```

Invocata con p=n [alvie]

Esempi

- Altro esempio interessante, utilizzato ampiamente nella teoria della complessità:
  - *Soddisfacibilità di formule booleane*

Definizioni

- Insieme V di variabili Booleane
  - Letterale: variabile o sua negazione
  - Clausola: disgiunzione di letterali
- Una espressione Booleana su V si dice in forma normale congiuntiva (FNC) se è espressa come congiunzione di clausole
- Una formula Booleana quantificata è una espressione in FNC preceduta da una sequenza di quantificatori universali ed esistenziali (∀,∃) che legano ***tutte*** le variabili

SAT

- Data una espressione in *forma normale congiuntiva*, il problema della soddisfacibilità (SAT) richiede di verificare se esiste una assegnazione di valori di verità alle variabili che rende l'espressione vera

- Il problema delle formule Booleane quantificate richiede invece di verificare se una certa formula Booleana quantificata è vera

Algoritmo per formule quantificate

- Algoritmo ricorsivo
- Assegna valore 0 alla prima variabile e risolvi il resto. Poni in v0 il risultato
- Assegna valore 1 alla prima variabile e risolvi il resto. Poni in v1 il risultato
- Se il quantificatore della prima variabile è ∃, allora restituisci v0 ∨ v1
- Se il quantificatore della prima variabile è ∀, allora restituisci v0∧ v1

Algoritmo per SAT

- Simile al precedente
  - Si immaginano tutte le variabili *quantificate esistenzialmente* e si restituisce sempre v0 ∨ v1

- Se n è il numero delle variabili, quante diverse configurazioni esaminano gli algoritmi?

Algoritmo per SAT

- Simile al precedente
  - Si immaginano tutte le variabili *quantificate esistenzialmente* e si restituisce sempre v0 ∨ v1
- Se n è il numero delle variabili, quante diverse configurazioni esaminanogli algoritmi?
  - 2^n
  - Entrambi i problemi sono in ExpTime
  - Sono in Pspace?

Algoritmo per SAT

- Simile al precedente
  - Si immaginano tutte le variabili *quantificate esistenzialmente* e si restituisce sempre v0 ∨ v1
- Se n è il numero delle variabili, quante diverse configurazioni esaminanogli algoritmi?
  - 2^n
  - Entrambi i problemi sono in ExpTime
  - Sono in PSpace? Sì → PSpace ⊑ ExpTime

Algoritmo per SAT

- Simile al precedente
  - Si immaginano tutte le variabili *quantificate esistenzialmente* e si restituisce sempre v0 ∨ v1
- Se n è il numero delle variabili, quante diverse configurazioni esaminanogli algoritmi?
  - 2^n
  - Entrambi i problemi sono in ExpTime
  - Sono in Pspace? Sì → PSpace ⊑ ExpTime

Algoritmi polinomiali non noti!

Certificato

- In un problema decisionale siamo interessati a verificare se una istanza del problema soddisfa una certa proprietà
- Spesso, in caso di risposta affermativa, oltre alla semplice risposta (binaria) si richiede di fornire anche un oggetto y , dipendente dall'istanza x e dal problema, che possa *certificare* il fatto che x soddisfa la proprietà, giustificando quindi la risposta
  - y viene detto *certificato*

Certificato

- Certificato per SAT?

Certificato

- Certificato per SAT?
  - Un’assegnazione di verità alle variabili che renda vera l’espressione
- Certificato per la connettività di un grafo?

Certificato

- Certificato per SAT?
  - Un’assegnazione di verità alle variabili che renda vera l’espressione
- Certificato per la connettività di un grafo?
  - Un suo albero di copertura
- Certificato per il problema delle formule Booleane quantificate?

Certificato

- Certificato per SAT?
  - Un’assegnazione di verità alle variabili che renda vera l’espressione
- Certificato per la connettività di un grafo?
  - Un suo albero di copertura
- Certificato per il problema delle formule Booleane quantificate?
  - Un pò più complicato, vero?

Certificato

- Certificato per il problema delle formule Booleane quantificate?
  - Un pò più complicato, vero?
  - È sufficiente esibire un'assegnazione di valori di verità alle variabili?
  - Nel caso peggiore quante ne servono?

Certificato

- Certificato per il problema delle formule Booleane quantificate?
  - Un pò più complicato, vero?
  - Non è sufficiente esibire un'assegnazione di valori di verità alle variabili
  - Nel caso peggiore quante ne servono?
    - Numero esponenziale
  - In questo caso è difficile esprimere anche un certificato!

La classe NP

- Queste osservazioni suggeriscono di utilizzare il costo della verifica di una soluzione per caratterizzare la complessità del problema stesso
- *Informalmente: NP è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale*

Esempi

- Verificare SAT
  - Come?
- Verificare albero di copertura
  - Come?

Esempi

- Verificare SAT
  - Come?
- Verificare albero di copertura
  - Come?
- Non si può fare altrettanto con il problema delle formule Booleane quantificate
  - Attualmente non noto se tale problema sia in NP
    - Si congettura di no!
- Ma cosa vuol dire NP?
  - P sta per polinomiale, ma N?
    - Non vuol dire NON…

Non determinismo

- Negli algoritmi visti finora ogni passo è determinato univocamente dallo stato della computazione
  - Algoritmi deterministici
- Un algoritmo non deterministico, oltre alle normali istruzioni, può eseguire istruzioni del tipo

  Indovina $z \in \{0,1\}$
- Il valore di z influenza la prosecuzione della computazione

Esempio

- Un algoritmo non deterministico per SAT come potrebbe funzionare?

Esempio

- Un algoritmo non deterministico per SAT come potrebbe funzionare?
  - Indovina i valori da assegnare alle variabili e poi verifica (deterministicamente) la bontà dell'assegnazione fatta
  - Computazione descritta da un albero, dove le ramificazioni corrispondono alle scelte non deterministiche
    - Quella determistica è descritta da una catena
  - Quindi per SAT ***esiste*** almeno un cammino che porta a una foglia con valore 1

Esempio

- E un algoritmo non deterministico per il problema delle formule Booleane non quantificate?
  - Per le variabili esistenziali è facile: procediamo come con SAT
    - *Indoviniamo* il valore di queste variabili
  - Problema per le variabili universali

Esempio

- E un algoritmo non deterministico per il problema delle formule Booleane non quantificate?
  - Per le variabili esistenziali è facile: procediamo come con SAT
    - *Indoviniamo* il valore di queste variabili
  - Problema per le variabili universali
    - La formula deve essere vera per TUTTI i possibili assegnamenti a queste variabili
    - Quindi il non determinismo non aiuta affatto
      - Bisogna verificare che tutte le foglie di un intero sottoalbero (di dimensione esponenziale) siano

Esempio

- Quindi, abbiamo le stesse difficoltà incontrate per risolvere il problema stesso
  - Non riusciamo nemmeno a immaginare un certificato di dimensione polinomiale nella dimensione dell'istanza di ingresso

La classe NP

- Data una qualunque funzione f(n), chiamiamo NTime(f(n)) l'insiemi dei problemi decisionali che possono essere da un algoritmo *non deterministico* in tempo O(f(n))

- La classe NP è la classe dei problemi risolvibili in tempo *polinomiale non deterministico* nella dimensione n dell'istanza di ingresso:

$$NP=\cup_{c=0..\infty}NTime(n^c)$$

La classe NP: definizioni

- *Informalmente: NP è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale (deterministico)*
- La classe NP è la classe dei problemi risolvibili in tempo *polinomiale non deterministico* nella dimensione n dell'istanza di ingresso:

$$NP=\cup_{c=0..\infty} NTime(n^c)$$

La classe NP: definizioni

- *Informalmente: NP è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale (deterministico)*
- La classe NP è la classe dei problemi risolvibili in tempo *polinomiale non deterministico* nella dimensione n dell'istanza di ingresso:

$$NP=\cup_{c=0..\infty} NTime(n^c)$$

Relazione fra le due definizioni

Ogni algoritmo non deterministico può essere articolato in due fasi:

1. Non deterministica di costruzione del certificato
2. Deterministica di verifica del certificato

Gerarchia delle classi

- P è incluso in NP oppure no?

Gerarchia delle classi

- P è incluso in NP oppure no?
  - Ovviamente sì!
    - Un algoritmo deterministico è un caso particolare di un algoritmo non deterministico, in cui l'istruzione *indovina* non è mai usata
    - Visione diversa: ogni problema in P ammette un certificato verificabile in tempo polinomiale….come mai?

Gerarchia delle classi

- P è incluso in NP oppure no?
  - Ovviamente sì!
    - Un algoritmo deterministico è un caso particolare di un algoritmo non deterministico, in cui l'istruzione *indovina* non è mai usata
    - Visione diversa: ogni problema in P ammette un certificato verificabile in tempo polinomiale….come mai?
      - Eseguo l'algoritmo che risolve il problema per costruire il certificato!

Gerarchia delle classi

- NP è incluso in PSpace oppure no?

Gerarchia delle classi

- NP è incluso in PSpace oppure no?
  - Ovviamente sì!
    - La fase deterministica di verifica può essere condotta in tempo polinomiale solo se il certificato ha dimensione polinomiale!

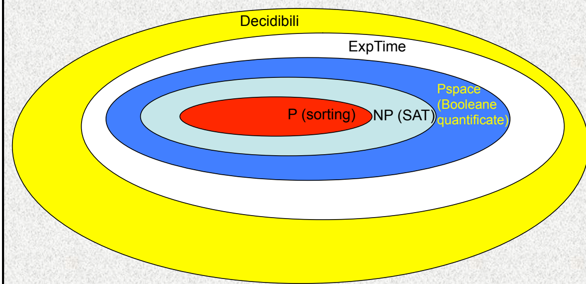
Gerarchia delle classi

- Quindi abbiamo

  $P \sqsubseteq NP \sqsubseteq Pspace \sqsubseteq ExpTime$

- Si congettura inoltre che le inclusioni siano proprie
  - Nessuno è finora riuscito a dimostrarlo
  - I problemi che si ritiene appartengano a NP ma non a P si dicono NP-completi
  - Si ritiene che il problema delle formule Booleane quantificate (che appartiene a PSpace) non appartenga a NP

Gerarchia delle classi



SUDOKU

- Tabella 9 X 9 contenente numeri in [1..9]
- Divisa in 3 X 3 sottotabelle (di taglia 3 X 3)
- Alcune celle contengono numeri, altre vuote
- Riempire le celle vuote in modo che
  1. ogni riga contenga una permutazione di 1,2,...,9
  2. ogni colonna contenga una perm. di 1,2,...,9
  3. ogni sottotabella contenga una perm. di 1,2,...,9

SUDOKU

3	9							8
	7	1			3			
		8		4	9		6	
1			2	7				9
6								3
5				3	6			4
	4		1	5		9		
			9			8	2	
9							4	7

3	9	6	5	1	2	4	7	8
4	7	1	6	8	3	5	9	2
2	5	8	7	4	9	3	6	1
1	3	4	2	7	5	6	8	9
6	8	7	4	9	1	2	5	3
5	2	9	8	3	6	7	1	4
8	4	2	1	5	7	9	3	6
7	1	3	9	6	4	8	2	5
9	6	5	3	2	8	1	4	7

- Soluzione ottenibile in questo caso attraverso implicazioni logiche
- Es. Sottotabella in alto a destra: il 3 può stare solo qui

		8
	6	

BACKTRACK con scelte non uniche

			6		2		9	
								6
			7	3	1	5		8
4		9	3			6		5
		3				1		
5		8			7	9		2
		1	5	2	3			
7								
	6	2	9		4			

			6		2		9	
			8					6
			7	3	1	5		8
4	2	9	3	1	8	6	7	5
6	7	3	2			1	8	4
5	1	8	4	6	7	9	3	2
		1	5	2	3			
7			1	8	6			
	6	2	9	7	4			

Partendo dalla configurazione a sinistra, giungiamo nella configurazione a destra che ammette diverse scelte per ogni casella [alvie]

Backtrack: *algoritmo che esplora tali scelte, annullando gli effetti nel caso che non conducano a soluzione*

Backtrack per SUDOKU

- Esamina le m caselle vuote nell'ordine indicato da **PrimaVuota**, **SuccVuota**, **UltimaVuota**

```
1  Sudoku( casella ):                                        ⟨pre: casella vuota⟩
2    elenco = insieme delle cifre ammissibili per casella;
3    FOR (i = 0; i < |elenco|; i = i+1) {
4      Assegna( casella, elenco[i] );
5      IF (!UltimaVuota(casella) && !Sudoku(SuccVuota(casella))) {
6        Svuota( casella );
7      } ELSE {
8        RETURN TRUE;
9      }
0    }
1    RETURN FALSE;
```

Invocata con **casella = PrimaVuota()**

- Nel caso pessimo, esplora circa 9^m scelte ($m \leq 9^2$)
- In generale, tabella n X n: circa $n^m \leq n^{n^2} = 2^{n^2 \log n}$

SUDOKU: quale complessità?

- L'algoritmo di backtrack è quindi esponenziale ...ma il SUDOKU n X n è trattabile o meno?
- Dipende dall'esistenza di un algoritmo polinomiale: ad oggi, tale algoritmo è ignoto
- SUDOKU sembra avere una natura computazionale diversa da quella delle Torri di Hanoi:
  - *Torri di Hanoi: esiste dimostrazione formale che non si può risolvere in meno di 2^n-1 mosse*
  - *SUDOKU: è possibile verificare la correttezza di una data soluzione in tempo polinomiale* (cosa non possibile con Torri di Hanoi).

SUDOKU: verifica polinomiale della correttezza di una soluzione

```
1  VerificaSudoku( sequenza ):          ⟨pre: sequenza di m cifre, con 0 < m ≤ n²⟩
2    casella = PrimaVuota( );
3    FOR (i = 0; i < m; i = i+1) {
4      cifra = sequenza[i];
5      IF (cifra appare in casella.riga) RETURN FALSE;
6      IF (cifra appare in casella.colonna) RETURN FALSE;
7      IF (cifra appare in casella.sotto_tabella) RETURN FALSE;
8      Assegna( casella, cifra );
9      casella = SuccVuota(casella);
10   }
11   RETURN TRUE;
```

- Richiede circa m X n $\leq n^3$ passi (ordine di crescita)
- SUDOKU è in quale classe computazionale?

SUDOKU: verifica polinomiale della correttezza di una soluzione

```
1  VerificaSudoku( sequenza ):          ⟨pre: sequenza di m cifre, con 0 < m ≤ n²⟩
2    casella = PrimaVuota( );
3    FOR (i = 0; i < m; i = i+1) {
4      cifra = sequenza[i];
5      IF (cifra appare in casella.riga) RETURN FALSE;
6      IF (cifra appare in casella.colonna) RETURN FALSE;
7      IF (cifra appare in casella.sotto_tabella) RETURN FALSE;
8      Assegna( casella, cifra );
9      casella = SuccVuota(casella);
10   }
11   RETURN TRUE;
```

- Richiede circa m X n $\leq n^3$ passi (ordine di crescita)
- SUDOKU è in quale classe computazionale?
  - NP!
  - Questo problema è anche NP-completo
    - Non è noto alcun algoritmo polinomiale e non si è dimostrato che non esista

Problemi NP-completi

- Caratterizzano i problemi più difficili all'interno della classe NP
  - Se esistesse un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti I problemi in NP potrebbero essere risolti in tempo polinomiale, e P=NP
  - Quindi: o tutti i problemi NP-completi sono risolvibili deterministicamente in tempo polinomiale o nessuno di essi lo è

Riducibilità polinomiale

- Dati due problemi decisionali
  P1 ⊑ I1 x {0,1} e P2 ⊑ I2 x {0,1}

diremo che P1 è riducibile polinomialmente a P2 (P1 < P2) se esiste una funzione
f: I1 → I2 tale che

- f è calcolabile in tempo polinomiale
- Per ogni istanza *x* di P1 ed ogni soluzione *s* ∈ {0,1}
  $(x,s) \in P1 \Leftrightarrow (f(x),s) \in P2$

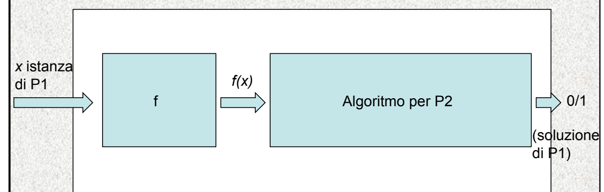
Riducibilità polinomiale

$(x,s) \in P1 \Leftrightarrow (f(x),s) \in P2$

- In altri termini, f trasforma un'istanza di P1 in un'istanza di P2 in modo tale che istanze positive di P1 risultino in istanze positive di P2 ed istanze negative di P1 risultino in istanze negative di P2

Riducibilità polinomiale

- Quindi, se esistesse un algoritmo per risolvere P2, potremmo utilizzarlo per risolvere P1



Esempio

- SAT (P1) è riducibile a quello delle formule Booleane quantificate (P2)
  - Sia e(x1, x2, …., xn) la formula SAT
  - La si trasforma in

    $\exists$ x1 $\exists$ x2 …. $\exists$ xn: e(x1, x2, …., xn)

Riducibilità polinomiale

- Risulta ovviamente che

  Se $P1 < P2$ e $P2 \in P$, *allora* $P1 \in P$

- Nota:la definizione di riducibilità polinomiale resta valida anche se la funzione f viene invocata un numero polinomiale di volte (e non solo 1 volta, come indicato nella definizione appena data)

Problemi NP-ardui e NP-completi

- Un problema decisionale P1 si dice *NP-arduo* se ogni problema Q $\in$ NP è riducibile polinomialmente a P1

- Un problema decisionale P1 si dice *NP-completo* se appartiene a NP ed è NP-arduo

Problemi NP-ardui e NP-completi

- Un problema decisionale P1 si dice *NP-arduo* se ogni problema Q $\in$ NP è riducibile polinomialmente a P1

- Un problema decisionale P1 si dice *NP-completo* se appartiene a NP ed è NP-arduo

Th. Se un qualunque problema decisionale NP-completo appartenesse a P, allora P=NP

NP-completezza

- Dimostrare che un problema è in NP può essere facile
  - Esibire un certificato polinomiale
- Risulta evidente però che non è altrettanto facile dimostrare che un problema P1 è NP-arduo
  - Bisogna dimostrare che TUTTI i problemi in NP si riducono polinomialmente a P1!
  - In realtà la prima dimostrazione di NP-completezza aggira il problema

Teorema di Cook-Levine

Th. Il problema SAT è NP-completo

- Idea: Cook ha mostrato un algoritmo che, dati un qualunque problema P ed una qualunque istanza *x* per P, costruisce una espressione Booleana in forma normale congiuntiva che descrive il calcolo di un algoritmo per risolvere P su *x*
- L'espressione è vera se e solo se l'algoritmo restituisce 1

Dimostrazioni di NP-completezza

- Sfruttano la transitività delle riduzione polinomiale

Se P1<P2 e P2<P3, allora P1<P3

Problema della clique

- Una *clique* di dimensione t è un grafo di t vertici che sono tutti connessi da coppie di archi
- Il problema della *clique* richiede di verificare, dati un grafo G ed un intero k, se G contiene una clique di dimensione almeno k

Th. Il problema della clique è NP-completo

Clique

- È facile vedere che il problema della clique è in NP
- Per dimostrare che è NP-arduo, mostriamo che 3-SAT si riduce ad esso
  - 3-SAT è una variante di SAT, dove ogni clausola è formata da 3 letterali. Anche 3-SAT è NP-completo

Clique

- Consideriamo una espressione Booleana *e* in forma normale congiuntiva: sia k il numero di clausole e siano l1i, l2i e l3i i 3 letterali nella i-esima clausola ci
- Costruiamo da e un grafo G come segue
  - Per ogni clausola, G contiene 3 vertici, uno per clausola
  - Due vertici sono connessi tra loro se i corrispondenti letterali sono in clausole diverse e non sono l'uno la negazione dell'altro
    - (x1 ∨ x2 ∨ x3) ∧ (!x1 ∨ !x2 ∨ !x3) ∧ (!x1 ∨ x2 ∨ x3)

Clique

- Mostriamo che *e* è soddisfacibile se e solo se G contiene una clique di k vertici
- (**Suff.**) Se *e* è soddisfacibile, *almeno* un letterale in ciascuna clausola è true
- Sia *esattamente* un letterale a **true** per ogni clausola
- Osserviamo i vertici di G *corrispondenti* ai letterali scelti

Clique

- Osserviamo i vertici di G *corrispondenti* ai letterali scelti
  - Tali vertici sono tutti connessi a coppie
    - Corrispondono a letterali in clausole diverse che non sono l'uno la negazione dell'altro
      - Non potrebbero essere contemporaneamente veri!
  - Questa è dunque una clique in G di dimensione k!
- (**Neces.**) simile alla precedente

Altri famosi problemi NP-completi

- **Copertura di vertici**
  - Una copertura di vertici (vertex cover) di un grafo G=(V,E) è un insieme di vertici C ⊆ V tale che per ogni (u,v) ∈ E, almeno uno tra u e v appartiene a C
  - Bisogna verificare se, dati G e un intero k, esiste una copertura di vertici di G di dimensione al più k

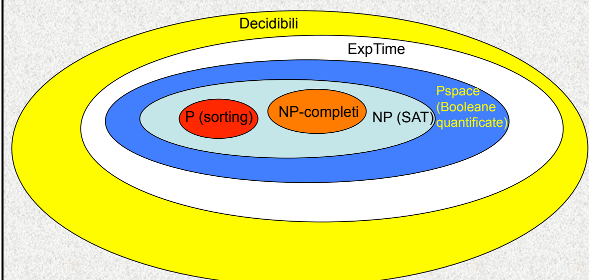
Altri famosi problemi NP-completi

- **Commesso viaggiatore**
  - Dati un grafo *completo* G con pesi w sugli archi ed un intero k, bisogna verificare se esiste un ciclo di peso al più k che attraversa ogni vertice una ed una sola volta
- **Colorazione**
  - Dati un grafo G ed un intero k, bisogna verificare se è possibile colorare i vertici di G con al più k colori tali che due vertici adiacenti non siano dello stesso colore

Altri famosi problemi NP-completi

- **Somme di sottoinsiemi**
  - Dati un insieme S di numeri naturali ed un intero t, bisogna verificare se esiste un sottoinsieme di S i cui elementi sommano esattamente a t
- **Zaino**
  - Dati un intero k, uno zaino di capacità c, e *n* oggetti di dimensioni s1, …., sn cui sono associati profitti p1, …., pn, bisogna verificare se esiste un sottoinsieme degli oggetti di dimensione <= c che garantisca profitto >=k

Gerarchia delle classi aggiornata



Algoritmi di ottimizzazione

- Abbiamo visto che un algoritmo di ottimizzazione può essere trasformato in un algoritmo di decisione che non è più difficile da risolvere del problema stesso
  - Minimo albero di copertura: dato un grafo G ed un intero k, esiste un albero di copertura di G di costo al più k?
- Cosa fare nel caso si renda necessario risolvere un problema di ottimizzazione la cui versione decisionale sia NP-completa?

Algoritmi di approssimazione

- A volte, avere una soluzione esatta non è strettamente necessario
- Una soluzione che non si discosti troppo da quella ottima potrebbe comunque risultare utile
  - Ed è magari più semplice da calcolare
- La teoria dell'approssimazione formalizza questa semplice idea

Algoritmi di approssimazione

- In generale esistono numerose soluzioni ammissibili, ma non tutte sono ottime
  - Problema della colorazione di un grafo
  - Soluzione approssimata?

Algoritmi di approssimazione

- In generale esistono numerose soluzioni ammissibili, ma non tutte sono ottime
  - Problema della colorazione di un grafo
  - Soluzione approssimata: colorare ogni vertice con un colore diverso
  - Soluzione ottima solo se il grafo è completo

Algoritmi di approssimazione

- Sia P1 un problema di ottimizzazione
- Un algoritmo di approssimazione per P1 ha fattore di approssimazione r(n) se il costo C della soluzione prodotta dall'algoritmo su una qualunque istanza di dimensione n differisce di un fattore moltiplicativo <= r(n) dal costo C* di una soluzione ottima

$$\max\{C/C^*, C^*/C\} \le r(n)$$

Algoritmi di approssimazione

$$\max\{C/C^*, C^*/C\} \le r(n)$$

- Questa formula definisce il fattore di approssimazione sia per problemi *massimizzazione* che di *minimizzazione*
- *Minimizzazione*: $C \ge C^* \rightarrow C \le C^* r(n)$
- *Massimizzazione*: $C \le C^* \rightarrow C \ge C^* r(n)$
  - Se r(n)=1, C è chiaramente una soluzione ottima
  - C sarà tanto peggiore quanto più si discosta da C*

FINE