

# **Elementi di complessità computazionale**

Prof. Paolo Brandimarte

Dip. di Scienze Matematiche – Politecnico di Torino

e-mail: [paolo.brandimarte@polito.it](mailto:paolo.brandimarte@polito.it)

URL: [staff.polito.it/paolo.brandimarte](http://staff.polito.it/paolo.brandimarte)

Questa versione: 26 febbraio 2024

**NOTA:** A uso didattico interno per il corso di laurea in Ingegneria Matematica PoliTO. Da non postare o ridistribuire.

## Contenuto

---

Le slide seguenti sono tratte dal capitolo 9 di: P. Brandimarte, *Ottimizzazione per la Ricerca Operativa*, CLUT 2022.

- Complessità di problemi e algoritmi. → Differenza tra la complessità
- Teoria della NP-completezza: classi  $\mathcal{P}$ ,  $\mathcal{NPC}$  e  $\mathcal{NPH}$ .

bene vs mal condizionamento  
i.e. sorting  
↑  
problema : complessità intrinseca  
del problema  
algoritmi indipendentemente dagli algoritmi  
↓  
i.e. bubble sort, quick sort  
stabile vs instabile

Quanto è complicato risolvere il problema in relazione alla dimensione del problema.

## Complessità di problemi e algoritmi

Si vuole caratterizzare la complessità in funzione della dimensione di un problema.  
Occorre distinguere la complessità degli algoritmi da quella dei problemi.

Nel caso di un algoritmo caratterizzato da un numero finito di passi, possiamo valutare (eventualmente nel caso peggiore) il numero di operazioni elementari in funzione della dimensione  $n$  del problema.

**NB.** Non saremo molto precisi nel caratterizzare la dimensione del problema (spazio di memoria richiesto) e il numero di passi (su macchina di Turing). Inoltre ci limitiamo a trattare la complessità nel caso peggiore (non quello medio).

Per esempio, *Perche' il numero di permutazioni e'  $n!$  → Es. Comesso viaggiatore che vuole trovare il ciclo Hamiltoniano* un algoritmo che consideri tutte le permutazioni di  $n$  oggetti ha complessità  $O(n!)$  ed è certamente non pratico per  $n$  grandi. Allo stesso modo, un algoritmo che considera tutti gli assegnamenti possibili di  $n$  variabili binarie ha complessità esponenziale  $O(2^n)$ . → Complessità esponenziale

*Rappresenta il caso peggiore in relazione allo spazio di memoria utilizzata*  
Al contrario, un algoritmo di ordinamento di  $n$  oggetti ha complessità polinomiale. Gli algoritmi più semplici hanno complessità  $O(n^2)$ , altri hanno complessità  $O(n \log_2 n)$ .

*algoritmi per ordinare gli elementi di un vettore*    *Complessità polinomiale*    *Complessità migliore*  
Nel caso di algoritmi iterativi che generano una sequenza di soluzioni, si può cercare di caratterizzare la velocità di convergenza (es., lineare o quadratica).

Immaginiamo una macchina e dover mettere i peni in sequenza di lavorazione da consegnare  $\Rightarrow$  trovare la sequenza che minimizza il ritardo di consegna  $\Rightarrow$  ordiniamo  $O(n^2)$   
 Ma se posso elaborare a partire da un tempo  $x$  che non è oggi? Serve un algoritmo branch and bound che ha complessità esponenziale e ciò non va molto bene! E' il problema che vieta c. minori? o no?

Una questione più sottile si pone quando si vuole caratterizzare la complessità intrinseca di un problema.  $\rightarrow$  A prescindere dall'algoritmo

• Un algoritmo è stabile o no (converge alla sol.) mentre un problema può essere bene o male condizionato  $\rightarrow$  è una caratteristica intrinseca del problema

Problema di sequenziamento Per comprendere la natura della questione, consideriamo un semplice problema di scheduling di  $n$  job su macchina singola. Indichiamo con  $p_j$ , il tempo necessario per il job  $j = 1, \dots, n$ , e con  $d_j$ ,  $j = 1, \dots, n$  la sua data di consegna (due date).  
 → Abbiamo una macchina che deve eseguire qualche compito una volta  $\Rightarrow$  Vogliamo trovare una sequenza di compiti da svolgere  
 La soluzione è una sequenza di job, ovvero una permutazione  $\sigma$  in cui  $\sigma(k)$  è l'indice del job in posizione  $k$ . I tempi di completamento sono

Non consideriamo i tempi di setup e consideriamo di poter fare tutti i job a partire da ora

$$C_{\sigma(1)} = p_{\sigma(1)}, \quad \text{tempo speso per eseguire il compito}$$

$$C_{\sigma(k)} = C_{\sigma(k-1)} + p_{\sigma(k)}, \quad k = 2, \dots, n.$$

Così  $\sigma(1)$  è il primo job da eseguire  
 $\sigma(1) \ \sigma(2) \dots$

Si vuole minimizzare la massima lateness,  $\rightarrow$  Vogliamo minimizzare i ritardi

Non ha senso sommare le lateness

- Si perdono le date di scadenza che diventano costanti
- Dalla media non guadri nulla

NOTA. Con  $L_j$  si intende la lateness, quindi rappresenta la distanza tra il tempo di completamento e la "data di scadenza"

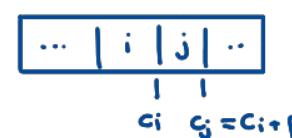
dove  $L_j \doteq C_j - d_j$ . Tale problema viene indicato con la stringa  $1//L_{\max}$ .  $\rightarrow$  È un problema di classe

Faccio un problema di min-max, considero quindi il ritardo nel caso peggiore  $\rightarrow$  Assomiglia un po' alla norma L<sub>∞</sub> tempo a cui ho completato il job j - la data di scadenza > o ritardo < o anticipo

IDEA. Ordiniamo i job in base alle date di consegna  $\Rightarrow$  Passo prima quella con la data più prossima. Per il problema  $1//L_{\max}$  esiste una soluzione ottima in cui  $d_{\sigma(k)} \leq d_{\sigma(k+1)}$ .  $\rightarrow$  Ordinando il vettore ho un algoritmo polinomiale: non così male  
 ↳ Otteniamo quindi la soluzione ottima. Dimostriamolo. Il problema è polinomiale

\* **Dimostrazione.** Supponiamo che esista una soluzione ottima in cui, per due job consecutivi in sequenza, prima  $i = \sigma(k)$  e poi  $j = \sigma(k+1)$ , si abbia  $d_i > d_j$ . Indichiamo con  $C_i$  e  $C_j = C_i + p_j$  i due tempi di completamento nella soluzione corrente, per la quale abbiamo due valori di lateness  $L_i = C_i - d_i$  e  $L_j = C_j - d_j$ .

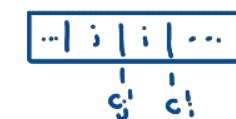
tempo di completamento quando finisco tempo speso per completare il job L'idea è che sto partendo dalla considerazione di avere  $d_i > d_j$ , ma di servire i prima di j. Quindi pongo i con data di scadenza maggiore,  
 \* Per assurdo, supponiamo che la regola non valga da eseguire prima rispetto a j



NB Se scambi i job « $i$ » e « $j$ » e la lateness migliora arriviamo alla contraddizione perché questo NON potrà essere una sol. ottima

Scambiamo allora  $i = j$

Quindi  
seguendo  
la EDD <->



Questo vuol dire che stiamo facendo eseguire  $j$  prima e  $i$  dopo, li stiamo proprio scambiando

Se scambiano i due job, avremo  $C'_j$  e  $C'_i = C_j$ . Per il job  $j$ , che anticipiamo, abbiamo  $C'_j < C_j$  e  $L'_j < L_j$ , e quindi la lateness del job  $j$  non può che migliorare nella nuova soluzione. Per il job  $i$ , che viene spostato in avanti nella sequenza, abbiamo

$j$  lo migliori sicuro perchè lo eseguo prima, quindi la distanza tra il tempo di completamento e la data di scadenza diminuisce

$$L'_i = C'_i - d_i = C_j - d_i < C_j - d_j = L_j, \rightarrow \text{ma allora la soluzione migliora}$$

$\checkmark C'_i = C_j$  perché non ho setup e robe varie  
 $d_i > d_j$

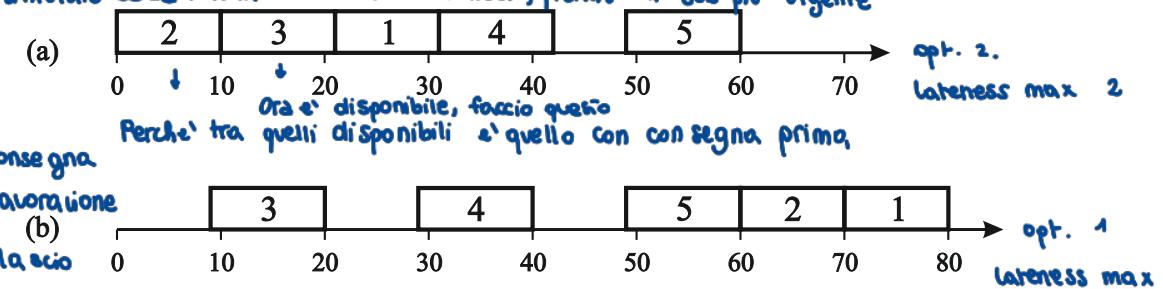
quindi il job  $i$  peggiora la sua lateness, che però non potrà essere peggio della vecchia lateness del job  $j$ . La nuova soluzione migliora quella precedente, contraddicendo l'assunzione di ottimalità.

Abbiamo quindi un algoritmo di complessità polinomiale per il problema. Ma cosa accade se complichiamo leggermente il problema, introducendo dei tempi di rilascio (*release time* o *ready time*)  $r_i$ ,  $i \in [n]$ , dei job? Per il problema  $1/r_i/L_{\max}$  non sono noti algoritmi di complessità polinomiale, ed è facile costruire controesempi alla regola EDD (da adattare comunque alla disponibilità di job).

↳ Applicabile in 2 modi / 1. Ordina sulla data di consegna e guarda quanti saranno i tempi di completamento reali / 2. Immagiamo di simulare l'ordine. Quando il servente è libero, prende il job più urgente

FACCIAMO UN CONTROESEMPIO ALLA EDD

$J_i$	$J_1$	$J_2$	$J_3$	$J_4$	$J_5$	Job
$d_i$	80	70	20	40	60	Data di consegna
$p_i$	10	10	11	11	11	Tempi di lavorazione
$r_i$	0	0	9	29	49	Tempi di rilascio



NB Ragionare sull'urgenza della data di consegna non dà la sicurezza della convergenza della soluzione ottima.  
Fornisce sol' ottima

Non per il modello misto intero

Esistono algoritmi branch-and-bound per il problema  $1/r_i/L_{\max}$  (quindi complessità esponenziale). Non si conosce un algoritmo di complessità polinomiale per questo problema di ottimizzazione combinatoria (e per molti altri), ma neppure è stato dimostrato che esso non possa esistere.

⚠  
Nel caso peggiore ha complessità esponenziale

Vogliamo capire se il problema è intrinsecamente complicato e non può avere un algoritmo di complessità polinomiale

Per il nostro esempio non abbiamo dimostrato  $\leq_{\text{nc}}^{\text{n}^3}$  l'algoritmo  
 $\leq_{\text{nc}}^{\text{n}^3}$  il problema può far ottenere l'algoritmo giusto

## Caratterizzazione della complessità di problemi: classi $\mathcal{P}$ , $\mathcal{NPC}$ e $\mathcal{NPH}$

Notiamo che il commesso viaggiatore è equivalente a  $\sum_i |l_i| l_{\max}$ , cioè se troviamo un algo polinomiale per uno dei 2 dimostra l'altro anche x l'altro

Esiste una classe molto ampia di problemi di ottimizzazione per cui non sono disponibili algoritmi di complessità polinomiale.

Tuttavia, la questione dell'esistenza o meno di un algoritmo di complessità polinomiale per essi rimane aperta.

La teoria della NP-completatezza ci permette di dare una risposta parziale alla questione, mostrando come questi problemi siano equivalenti tra di loro, nel senso che un algoritmo di complessità polinomiale per anche uno solo di essi fornirebbe un algoritmo di complessità polinomiale per tutti i problemi di una classe molto ampia.

Non dimostrò che c'è oppure non c'è

Il fatto che decenni di ricerca sulla soluzione di tutti questi problemi non abbiano prodotto un algoritmo polinomiale suggerisce che esso in effetti non esiste.

Risposta sì o no  
Occorre distinguere **problemi di decisione** e **problemi di ottimizzazione**. Esempi di problema di decisione sono i seguenti:

Problema di decisione: non sto ottimizzando nulla, c'è solo la risposta sì/no

Problema  $K_0$  (subset sum): Dati  $n+1$  numeri interi positivi  $a_1, a_2, \dots, a_n$  e  $b$ , esiste

E' parente al problema dello zaino un sottoinsieme  $\mathcal{J} \subseteq [n]$  tale che  $\sum_{i \in \mathcal{J}} a_i = b$ ?  $\Rightarrow$  Voglio pagare senza dover ricevere indietro il resto

un sottoinsieme di  $a_1, a_2, \dots, a_n$

Problema equivalente. Partizioni per 2 somme equivalenti

Anche questo è un problema di decisione Problema  $LS_0$ : Dato un problema di lot sizing multiprodotto, con tempi e costi di setup, esiste una soluzione ammissibile rispetto al soddisfacimento della domanda e ai vincoli di capacità produttiva?  $\rightarrow$  La domanda è essere una soluzione!

La risposta è sì o no, non serve trovare effettivamente la soluzione nel caso in cui esistesse!

Data una specifica istanza di un problema di decisione, la risposta è sì oppure no.

Dal punto di vista teorico, è più agevole trattare problemi di decisione, ma è facile vedere il legame tra problemi di ottimizzazione e problemi di decisione. Dato un problema di ottimizzazione

Problema di soddisfacimento di vincoli

$$\min_{x \in S} f(x), \Rightarrow \text{Vogliamo capire quanto questo problema è intrinsecamente complicato}$$

possiamo definirne una versione decisionale, scegliendo un numero  $k$  e chiedendoci se esiste  $x \in S$  tale che  $f(x) \leq k$ , dove  $k$  è un numero intero. Indichiamo

Da ottimizzazione a decisione facendo un problema di decisione

- con PO il problema di ottimizzazione,  $\Rightarrow \min_{x \in S} f(x)$

$f(x) \leq k$  voglio un problema che soddisfa questi vincoli

- e con PD il corrispondente problema di decisione.  $\Rightarrow f(x) \leq k$  ammette soluzione

Se abbiamo a disposizione un algoritmo efficiente per PO, allora possiamo risolvere in modo efficiente anche PD: basta risolvere il problema di ottimizzazione e verificare se  $f(x^*) \leq k$ . Questo ci permette di scrivere

Se so risolvere l'ottimizzazione, so risolvere il problema di decisione

$\text{complessità (PD)} \leq \text{complessità (PO)} \Leftarrow \text{PD} \rightarrow \text{PO},$

Se ho un algo buono per PO, ce l'ho anche per PD

Se so risolvere PO, risolvendolo vedo se il problema PD è sì o no

nel senso che il problema di decisione può essere ricondotto al problema di ottimizzazione.

Non è detto che tale trasformazione sia conveniente, ma possiamo escludere che PO sia facile se PD è difficile, perché un ipotetico algoritmo efficiente per PO risolverebbe anche facilmente PD.  $\rightarrow$  L'ottimizzazione NON puo' essere piu' facile rispetto al problema di decisione

Se PD è complicato, anche PO lo sarà

Quindi, per dimostrare che un problema di ottimizzazione è difficile, può bastare dimostrare che è difficile il corrispondente problema di decisione.

D'altro canto, un algoritmo di decisione efficiente potrebbe, in certi casi, essere utilizzato per risolvere il problema di ottimizzazione. Se la funzione di costo in PO assume valori interi non negativi, e abbiamo un upper bound  $U$  sul costo ottimo, possiamo applicare una procedura di bisezione.

con uneuristica lo possiamo trovare  
basta riscalare il valore



Indichiamo con  $\mathcal{P}$  la classe dei problemi di decisione per cui esiste un algoritmo di complessità polinomiale, in grado di risolvere tutte le possibili istanze del problema.

→ Sai risolvere il problema e oltre se la risposta è sì o no (basta valutare se fa corretto)

Con questo vogliamo dire che il numero di passi, e quindi la complessità temporale dell'algoritmo è limitata superiormente da una funzione polinomiale dello spazio di memoria necessario per descrivere ogni istanza del problema.

Un tale algoritmo è in grado di fare due cose:

1. generare una soluzione; → è difficile

2. verificarne la correttezza.

Ragioniamo sui problemi di decisione:  
Classe  $\mathcal{P}$  - Quanto tempo e spazio impegni per risolvere questo problema è una funzione limitata superiormente da un polinomio

e' computazionalmente molto semplice

→ Esiste effettivamente la soluzione che soddisfa i vincoli

→ esiste un certificato polinomiale, cioè esiste un modo veloce per verificare che la soluzione va bene

Esistono problemi, come  $K_0$ , per cui la prima parte del compito è difficile, ma la seconda no. Se enumeriamo, mediante un albero di ricerca, tutti i possibili sottoinsiemi  $\mathcal{J}$ , possiamo verificare se una specifica istanza ha risposta positiva o negativa, ma chiaramente tale algoritmo ha complessità esponenziale.

Se non sappiamo fare di meglio, con un albero di ricerca si possono enumerare tutti i sottoinsiemi e giungere a una soluzione ⇒ complessità esponenziale

Tuttavia, se disponessimo di un ipotetico calcolatore **non deterministico**, in grado di eseguire un numero infinito di processi di calcolo in parallelo, saremmo in grado di risolvere il problema in tempo polinomiale.

→ Può eseguire infiniti calcoli in parallelo

→ Hanno un certificato polinomiale e con un calcolatore non deterministico si esegue in tempo polinomiale => e' polinomiale su un calcolatore non deterministico  
**Definizione (classe NP)** Si definisce classe  $\mathcal{NP}$  l'insieme dei problemi di decisione le cui istanze che hanno risposta positiva sono verificabili in tempo polinomiale.

Per definizione,  $\mathcal{P} \subseteq \mathcal{NP}$ . Una questione meno ovvia è se vale  $\mathcal{P} \equiv \mathcal{NP}$  o  $\mathcal{P} \subset \mathcal{NP}$  in senso stretto.  
e' non deterministico polinomiale, cioè non e' polinomiale su un'architettura di calcolo non deterministico  
→ esiste un algoritmo di complessità polinomiale su di un processore

È ragionevole, da questo punto di vista, cercare di caratterizzare la sottoclasse dei problemi più difficili in  $\mathcal{NP}$ .

Cosa si intende che i problemi sono equivalenti? E' possibile passare da uno all'altro in tempo polinomiale

**Definizione (riduzione in tempo polinomiale)** Siano  $P$  e  $Q$  due problemi di decisione, per cui ogni istanza  $I_P$  di  $P$  può essere trasformata in tempo polinomiale in un'istanza  $I_Q$  di  $Q$  tale che  
→ Posso trasformare un problema in un altro  
→ Negativa e' un'altra storia

Anche a deve essere difficile

$I_P$  ha risposta positiva se e solo se  $I_Q$  ha risposta positiva.

Diremo che  $P$  è **riducibile in tempo polinomiale** a  $Q$ , e useremo la notazione  $P \prec Q$ . La notazione  $P \prec Q$  sottolinea che la complessità di  $P$  non è maggiore della complessità di trasformare  $P$  in  $Q$  e poi risolvere  $Q$ ,  
P e' difficile

$$\text{compl}(P) \leq \text{compl}(Q) + \text{compl}(P \rightarrow Q).$$

di risolvere Q  
di risolvere P  
Se la trasformazione ha complessità polinomiale  
questo è praticamente trascurabile  
⇒ Non è detto che trasformare P in Q sia una buona idea  
Complessità di trasformare P in Q

Se la trasformazione ha una complessità trascurabile, la riduzione di  $P$  a  $Q$  mostra che  $Q$  non è più facile di  $P$ . Se  $P$  è difficile e  $P \prec Q$ ,  $Q$  non può essere facile. Altrimenti, potremmo trasformare un'istanza di  $P$  in una di  $Q$ , e poi applicare l'algoritmo efficiente per  $Q$ .

NB Un problema di OTTIMIZZAZIONE puo' essere  
NP-hard, ma NON puo' essere NPC

$\omega^{\text{NP}}$  e' possibile aggiungere i problemi di ottimizzazione

Diamo la caratterizzazione dei problemi difficili

se e' difficile almeno quanto tutti i problemi di  $\omega^{\text{NP}}$ , e' ragionevole

**Definizione.** Un problema di decisione  $P$  è detto **NP-difficile (NP-hard)** se ogni problema nella classe  $\mathcal{NP}$  è riducibile a  $P$ . Indichiamo con  $\mathcal{NPH}$  la classe dei problemi NP-difficili.

Nota che  $P$  non è detto che appartenga a  $\omega^{\text{NP}}$

Se ogni problema di  $P$  è riconducibile a  $\omega^{\text{NPC}}$  che contiene solo problemi di decisione (non di ottimizzazione)  
**Definizione.** Un problema di decisione  $P$  è detto **NP-completo** se è in  $\mathcal{NP}$  ed è NP-difficile. Indichiamo con  $\mathcal{NPC}$  la classe dei problemi NP-completi.

e'  $\omega^{\text{PH}}$ , ma e' dentro  $\omega^{\text{P}}$ , quindi e' un problema di decisione

Le implicazioni pratiche di tali definizioni sono:

1. un problema NP-difficile non è più facile di un problema qualsiasi in  $\mathcal{NP}$ ;  
perche' si possono ricondurre tutti a questi  $\omega^{\text{PH}} \cap \omega^{\text{P}} = \omega^{\text{PC}}$
2. la classe  $\mathcal{NPC}$  è la classe dei problemi più difficili in  $\mathcal{NP}$ .

Per dimostrare che un problema di decisione  $P$  è NP-completo, occorre dimostrare:  
quindi che e' un problema di decisione complicato

1. che  $P$  è in  $\mathcal{NP}$ ;
2. che un problema NP-completo  $Q$  può essere ridotto in tempo polinomiale a  $P$ , cioè  $Q \prec P$ . → questa e' la parte complicata

Osserviamo che, dato che  $Q$  è NP-completo,  $P \prec Q$ , e se trascuriamo la complessità della trasformazione, questo implica

$$\text{compl}(P) \leq \text{compl}(Q).$$

Tutti i problemi di NP hanno la stessa complessità

Se trovo un algoritmo polinomiale per un problema di NP, li trovo tutti e andrei a dimostrare che  $P \in NP$

Ma il secondo passo della dimostrazione di NP-completezza, ovvero dimostrare che  $Q \leq P$ , implica anche che

$$\text{compl}(Q) \leq \text{compl}(P).$$

Le due diseguaglianze, sempre a meno della complessità della trasformazione, mostrano che

$$\text{compl}(Q) = \text{compl}(P).$$

→ Non tenendo in considerazione la complessità della trasformazione

In altre parole, i problemi della classe  $NPC$  sono equivalenti in termini di complessità computazionale, e un algoritmo polinomiale per uno di essi permetterebbe di risolverli tutti in tempo polinomiale.

Ricorda. Visto che NP contiene i problemi polinomiali su calcolatori non deterministici, allora conterrà anche i problemi di P che sono polinomiali su calcolatori deterministici (quindi limitati)

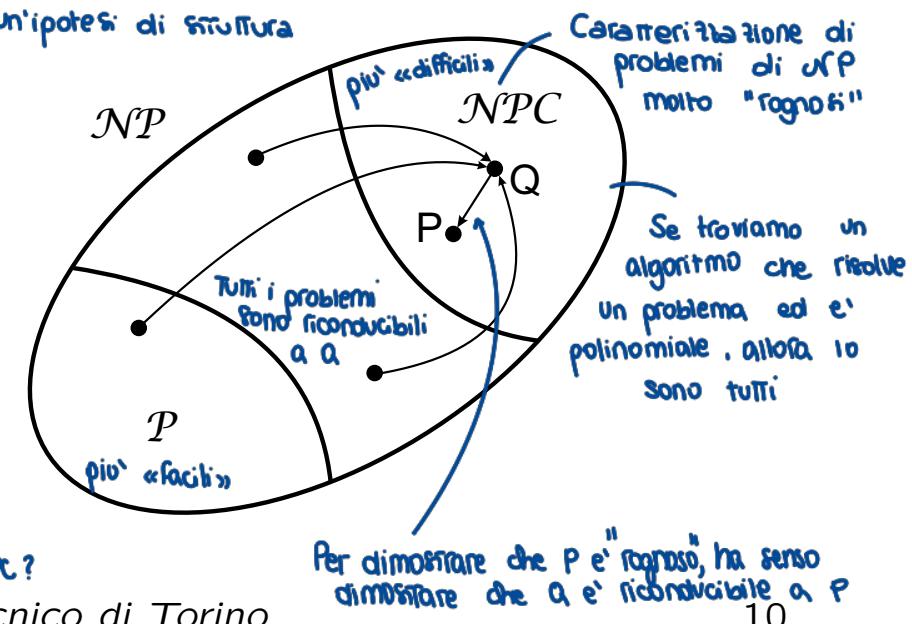
i. Avremmo quindi  $P = NP$ , ipotesi non troppo plausibile a causa dell'equivalenza di una vasta classe di problemi per i quali non è noto un algoritmo di complessità polinomiale, nonostante essi siano stati oggetto di ampio studio nel corso degli anni.

ii. Se accettiamo l'ipotesi  $P \neq NP$ , possiamo rappresentare le relazioni tra le classi  $P$ ,  $NP$  e  $NPC$  come in figura. Essa ipotizza una gerarchia per cui la classe  $P$  sarebbe la classe dei problemi più facili in  $NP$ , e  $NPC$  quella dei problemi più difficili.

Tutti i problemi in  $NP$  si possono trasformare nel problema  $Q \in NPC$ . Se  $P \in NP$ , per dimostrare che  $P \in NPC$ , occorre trasformare  $Q$  in  $P$ .

Ma come si trova l'innesco della catena? Com'è possibile trovare il primo problema di  $NPC$ ?

Paolo Brandimarte – Dip. di Scienze Matematiche, Politecnico di Torino



Problema di omimittazione puo' appartenere a  $\text{NP} \setminus \text{P}$ , ma non a  $\text{NP} \cap \text{P}$ !

Se dimostriamo che un problema  $P$  è in  $\text{NPC}$ , questo può a sua volta essere trasformato in altri problemi, permettendoci di ampliare la classe dei problemi noti in  $\text{NPC}$ . Il punto critico, chiaramente, è trovare l'innesto della catena, ovvero il primo problema in  $\text{NPC}$ , al quale tutti i problemi in  $\text{NP}$  possono essere ricondotti.

Da cui poi partono tutti gli altri

Il teorema di Cook dimostra che **problema della soddisfacibilità** soddisfa i requisiti necessari e ci fornisce la soluzione.

E' un problema  $\text{NP} \setminus \text{P}$ ! Allora a almeno un problema  $\text{NP} \setminus \text{P}$ !!

(or) and (or)

Data una formula Booleana in forma canonica disgiuntiva, decidere se esiste un'assegnamento di valori ai suoi elementi che la rende vera.

Per esempio, la formula

Con  $A, B, C \in \{\text{True}, \text{False}\}$

$B, C = V$   
e' soddisfacibile

Nelle clausole almeno una deve essere vera

$(A \text{ or } B) \text{ and } (\text{not}(A) \text{ or } C)$ ,  $\rightarrow$  Tutte le clausole devono essere vere

definita rispetto alle variabili Booleane  $A, B$  e  $C$ , è soddisfatta se  $B$  e  $C$  sono entrambe vere. Al contrario,

Non e' soddisfacibile  
perche' A e not(A) dovrebbero  
essere vere entrambe

$(A \text{ or } B) \text{ and } (\text{not}(A) \text{ or } B) \text{ and } (\text{not}(B))$

non può essere soddisfatta da alcun assegnamento di verità alle variabili.

A partire dal problema della soddisfacibilità, si può ricavare per riduzioni polinomiali successive una famiglia crescente di problemi NP-completi, compreso il problema  $K_0$ , che può essere considerato come un cugino in versione decisionale del problema knapsack.

e' riconducibile al problema  
della soddisfacibilità

Obiettivo. Il problema  $\text{Ko}$  è riconducibile al problema  $1/r_i/L_{\max}$

Dal problema  $\text{subset sum}$  cerco i dati per il problema di scheduling

Il fatto che tale problema faccia parte della classe  $\mathcal{NPC}$  ci permette di dimostrare il teorema seguente, che risolve la questione da cui siamo partiti.

Senza  $\text{r}_i$  abbiamo la soluzione in tempo polinomiale

**Teorema.** Consideriamo una versione decisionale del problema  $1/r_i/L_{\max}$ : dati i tempi di rilascio  $r_i$ , le date di consegna  $d_i$  e i tempi di lavorazione  $p_i$ , tutti a valori interi positivi, per  $n$  job  $J_i$ ,  $i \in [n]$ , esiste una soluzione in cui nessun job è completato in ritardo? Tale problema di decisione è NP-completo.

Domanda. Esiste una soluzione per cui tutti i job rispettano la data di consegna?

**Dimostrazione.** Il problema è chiaramente in  $\mathcal{NP}$ , poiché per una data sequenza è facile verificare se i job vengono completati in tempo rispetto alle due date.

Dati gli interi positivi  $a_j$ ,  $j \in [n]$ , creiamo  $n$  job  $J_j$  con parametri

$$r_j = 0, \quad p_j = a_j, \quad d_j = 1 + \sum_{k \in [n]} a_k, \quad j \in [n].$$

Creiamo un ulteriore job  $J_0$  con parametri

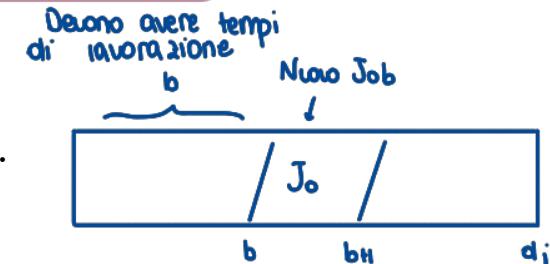
$$r_0 = b, \quad p_0 = 1, \quad d_0 = b + 1.$$

Perché tutti i job rispettino la data di consegna, è necessario che il job  $J_0$  inizi la lavorazione al tempo  $t = b$ . Inoltre, dato che la data di consegna degli altri job è pari alla somma di tutti i tempi di lavorazione, la soluzione non può presentare periodi di tempo in cui la macchina è ferma, prima di avere completato l'intero insieme di job.

Questo richiede che sia possibile individuare un sottoinsieme  $\mathcal{J}$  di job da schedulare prima di  $J_0$ , in modo tale che

$$\sum_{j \in \mathcal{J}} p_j = r_0.$$

Tale insieme risolve il problema subset-sum.



Per non avere job in ritardo la macchina non deve riposarsi mai

## Problema di Partitioning → ricondotto al problema del polinomio

Consideriamo  $m$  interi positivi  $a_j$  con  $j=1, \dots, m$ . Esiste un sottoinsieme  $J \subseteq \{1, \dots, m\}$  t.c.  $\sum_{j \in J} a_j = \sum_{j \notin J} a_j$ ?

Risolvere il problema  $\min p(x)$  con  $p(x)$  polinomio e' **NP-hard**.

Se troviamo il min del polinomio risolviamo il problema.

↪ E' un problema difficile perché ottimizzazione non convessa.



Infatti i polinomi hanno questa forma.

Definiamo le variabili  $x_j = \begin{cases} 1 & \text{se } j \in J \\ 0 & \text{se } j \notin J \end{cases}$ , allora  $\sum_{j=1}^n a_j x_j = 0$  è minimo rispetto a  $x_j$ . → Dal vincolo discreto  $x_j \in \{0, 1\}$  passo al continuo  $x_j \in [0, 1]$

Quindi costruiamo il seguente problema di ottimizzazione

$$\min_{x \in \mathbb{R}^n} \left( \sum_{j=1}^n a_j x_j \right)^2 + \left( \sum_{j=1}^n (x_j^2 - 1)^2 \right)$$

$\downarrow$

$\sum_j x_j = 0$

la vorrei idealmente a 0

→ ottieniamo la minimizzazione di un polinomio di ordine 4

Abbiamo dimostrato che tutti i problemi NP si posso ricondurre

Problema di ottimizzazione  
NPH

Vincolo che rende il problema non convesso

Consideriamo la funzione di penalità  $h_j(x) = x_j^2 - 1$

→  $\sum_{j=1}^n h_j(x_j)$  in quanto voglio che ogni singolo addendo ( $h_j$ ) sia zero

Problema di programmazione lineare nel continuo:  $\min c^T x$  (\*) . e' un problema polinomiale?

$$\begin{aligned} \text{s.t. } & Ax = b \\ & x \geq 0 \end{aligned}$$

↓  
Si, algoritmo dell'ellissoidale e' polinomiale

Si, algoritmo di punti interni e' polinomiale → quindi (\*) e' un problema polinomiale

## Dai problemi di decisione ai problemi di ottimizzazione

---

Il teorema dimostra che un problema di decisione legato al problema di ottimizzazione  $1/r_i/L_{\max}$  è NP-completo, ma cosa possiamo dire del problema di ottimizzazione stesso?

Per definizione, la classe  $\text{NPC}$  contiene solo problemi di decisione.

Possiamo però estendere le classi  $\mathcal{P}$  e  $\text{NPH}$ , includendo in esse anche problemi di ottimizzazione.  
Non necessariamente di decisione

I problemi di ottimizzazione per cui è noto un algoritmo di complessità polinomiale stanno in  $\mathcal{P}$ .

Nella classe  $\text{NPH}$  possiamo includere problemi di ottimizzazione ai quali possiamo ricondurre un corrispondente problema di decisione.

Nel nostro caso, la versione decisionale di  $1/r_i/L_{\max}$  si riduce chiaramente al problema di ottimizzazione.

Inoltre, nella dimostrazione abbiamo assunto che i dati fossero numeri interi. Ma il problema a numeri interi può evidentemente essere ridotto al problema generale. Possiamo quindi affermare che il problema di scheduling  $1/r_i/L_{\max}$  è NP-difficile.

Se esiste un algoritmo polinomiale avrei risolto tutti i problemi di  $\text{NP-H}$  in tempo polinomiale

## L'impatto della codifica di un problema

Nella trattazione ci siamo limitati alle classi fondamentali e siamo stati piuttosto informali e imprecisi.

Non possiamo però fare a meno di considerare l'impatto del modo in cui si codifica un problema. → Ricorda che bisogna considerare la memoria spesa per quell'algoritmo

Sarebbe infatti errato, per esempio, associare a un problema knapsack una dimensione pari al numero di oggetti. La dimensione si riferisce a una codifica binaria che comprende tutti i dati del problema. Per mostrare la rilevanza di ciò, consideriamo un classico algoritmo di programmazione dinamica per la soluzione del problema knapsack:

E' il problema dello zaino  
ed e' NP-hard

In generale e'  
approssimata perché  
non e' sempre possibile  
scriverla in forma  
TABULARE

$$\begin{aligned} \max \quad & \sum_{k=1}^n v_k x_k \rightarrow \text{Sottoinsieme degli oggetti che prendiamo} \\ \text{s.t.} \quad & \sum_{k=1}^n w_k x_k \leq B \rightarrow \text{Budget di peso o di volume} \rightarrow \text{ci' cio' che mi aiuta a non essere miope} \\ & x_k \in \{0, 1\} \quad k = 1, \dots, n. \end{aligned}$$

Definiamo la funzione valore

→ Mettendo la dinamica, mettiamo la variabile di stato  
 $V_k(s) \doteq$  valore del sottoinsieme ottimale tra gli oggetti  $\{k, k+1, \dots, n\}$ ,

in questo caso la variabile di stato e' la capacità rimanente  
quando la capacità residua è  $s$ .

E' un buon algoritmo (pseudo-polinomiale), ma se  $B$  e' grande diventa molto lento

Riprendiamo quando parleremo di  
programmazione dinamica

→ Prendiamo le decisioni in maniera sequenziale (tempo)

La programmazione dinamica e' un principio che sta alla base del reinforcement learning

In sostanza, la funzione valore assume che siano già state fatte scelte di inserimento o meno degli oggetti da 1 a  $k - 1$ ; a valle di tale selezione, abbiamo una capacità residua  $s$ , e ci chiediamo come utilizzarla al meglio per le scelte rimanenti. Se i dati  $w_k$  e  $B$  del problema sono interi, lo sarà anche la capacità residua  $s$ .

Per risolvere il problema, ovvero trovare il valore di  $V_1(B)$ , si applica una relazione ricorsiva:

Equazione funzionale

0	1	2	...	N
?				
B				

$$V_k(s) = \begin{cases} V_{k+1}(s) & \xrightarrow{\text{Esplicito l'idea di andare all'indietro}} \text{per } 0 \leq s < w_k, \\ \max \{V_{k+1}(s), V_{k+1}(s - w_k) + v_k\} & \xrightarrow{\text{non metto}} \text{per } w_k \leq s \leq B. \end{cases} \quad (1)$$

Si tratta di una equazione funzionale con condizione terminale:



$$V_n(s) = \begin{cases} 0 & \text{per } 0 \leq s < w_n, \\ v_n & \text{per } w_n \leq s \leq B. \end{cases} \quad \text{Procediamo all'indietro} \quad (2)$$

Occorre tabulare tutte le funzioni  $V_k(s)$ ,  $k = 1, \dots, n$ , per valori interi di  $s$ , che assume valori nel range da 0 a  $B$ . Pertanto, tale algoritmo ha complessità  $O(nB)$ .

Assumiamola come numero intero  
Riempiamo la matrice colona per colonna  
Ha complessità polinomiale? NO.  
Guarda la capacità di memoria richiesta per risolvere il problema

Questo non implica che il problema knapsack abbia complessità polinomiale: per rappresentare il valore  $B$  in aritmetica binaria bastano  $\lceil \log_2 B \rceil$  bit.

Quindi l'algoritmo, rispetto a tale codifica binaria, ha complessità esponenziale. Se si utilizzasse un computer con una codifica unaria, l'algoritmo che abbiamo considerato avrebbe complessità polinomiale.

Quindi otteniamo che non e' polinomiale  
Se si utilizzasse la codifica unaria sarebbe di complessità polinomiale

Si dice infatti che un algoritmo di questo tipo è **pseudo-polinomiale**.

## Problema di LOT SIZING

→ Dimensionamento dei lotti.

- Consideriamo →
- $n$  prodotti
  - $T$  periodi
  - $m$  risorse
  - $d_{it}$   $i=1, \dots, n, t=1, \dots, T$  rappresenta la domanda deterministica
  - $z_{it}$  quanto si produce del prodotto  $i$  nel tempo  $t$

Ci sono →

- Vincoli di capacità
- Tempi di set-up

Ottieniamo quindi il problema →

$$\min \sum_i \sum_t h_i I_{it} + \sum_i \sum_t F_i S_{it} \quad \text{se produce } i \text{ al tempo } t$$

che è lineare come  
tutti i problemi di  
programmazione lineare

el la variabile di stato  
 costo di storage nel magazzino  
 Rappresenta il magazzino (livello)  
 costo di set-up

\* Il fatto che il magazzino NON sia in negativo, implica che si è sempre in grado di soddisfare la domanda

$I_{it} = I_{i,t-1} + z_{it} - d_{it}$   $V_{it}$  → equazione dinamica nella variabile di stato;  
 costo tempo di set-up → rappresenta il magazzino, cioè quanto meno lascio lo

$\sum_i (Rim_{it} + Rim_{it}) \leq D_m \quad \forall m, t$  → vincolo sulla capacità produttiva  
 che accoppia i problemi e li rende non decomponibili

$$I_{it}, z_{it} \geq 0$$

Grande errore

$z_{it} \leq M S_{it} \quad \forall i, t$  → se non paga il costo fisso di set-up  $z_{it}$  è fissa a 0

$$S_{it} \in \{0, 1\}$$

Sentì questo è decomponibile e il problema è polinomiale

Il problema diventa NP-H quando aggiungiamo i set-up ⇒ è risolvibile con il branch and bound.

(tempo esponenziale)

• euristica → soluzione deve essere ammissibile prima che ottima

• MRP: Material Requirements Planning → sistemi per gestione operativa

MRPI, ERP

Aggiungiamo anche i tempi di set-up

Recap - 1) Problema di programmazione lineare P

2) Problema NP-H

3) Problema NPH → diventa NPC se consideriamo il problema di decisione e non quello di deputazione

Domanda E' possibile soddisfare le domande?

→ Riportiamo a un problema di decisione

Ma allora  $\exists$  un set di variabili decisionali che soddisfa i vincoli? (•)  $\Rightarrow$  Problema di feasibility ( $\rightarrow$  cioè e' realizzabile, ammette una soluzione oppure no?)

Il problema così formulato e' NPC? Rilassato nel continuo  $S_{t+1} \in \mathbb{R}_{+}$  ottengo un problema lineare

Ma così facendo  $S_{t+1} = \frac{z_t - d_t}{h}$ , ma anche se  $S_{t+1} \geq 0$  il set-up lo paghi lo stesso! Quindi rilassando nel continuo  
Non vuol dire giusto!

sostituisco i costi di set-up, ma posso ottenere un soluzione ammissibile  $\Rightarrow$  Il problema e' P.

aggiungiamo i tempi di set-up (quindi le  $S_t$  sono anche dei vincoli!)

Ma In questo problema, anche il problema di decisione e' NPC

funzione di ricorso in programmazione stocastica

$$\text{Per il problema: } \min \sum_{t=1}^T (h I_t + s d_t) \\ z_t \leq h S_t \quad \forall t$$

$$\xrightarrow{I_t \in \mathbb{Z}_+} \text{value function. } V_t(I_{t+1}) = \min_{\substack{z_t \geq d_t - I_t \\ z_t \in \mathbb{N}}} s d_t + h(z_t + I_t - d_t) + V_{t+1}(I_{t+1}, z_t - d_t)$$

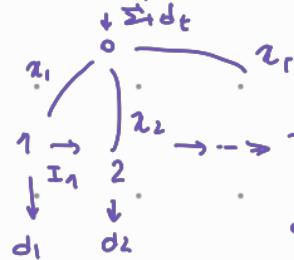
$$I_t = I_{t-1} + z_t - S_t \quad \forall t$$

$z_t, I_t \geq 0$  Soddisfacimento  
 $S_t \in \mathbb{N}$ , i.e. delle domande

E' pseudo-polinomiale. → al Knapsack

Se sfruttiamo la struttura del modello posso otteniamo algo polinomiale

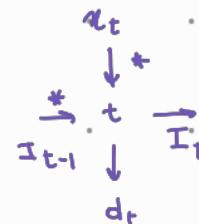
Pensiamo il problema come un flusso sulla rete. Supponendo  $I_{T+1}$  (senza perdita di generalità)



e consideriamo anche  $I_{T+1} = 0$ .

Quali sono i costi? Flusso di magazzino che paga solo se apri l'arco.

All'ultimo, vale la condizione  $I_{T-1}^* z_{T-1}^* = 0$



Non è giusto che i due archi (\*) siano entrambi ≠ 0



Riformulo il problema come percorso su grafo a costo minimo.  $\rightarrow$  è polinomiale SHORTEST PATH.

Ma il problema è che l'assunzione  $I_{T+1} = 0$  non va molto bene, inoltre questo modello funziona solamente per problemi statici e non dinamici.