Università Roma Tre – Dipartimento di Matematica e Fisica
IN440 – Ottimizzazione Combinatoria
Prof. Marco Liverani

Algoritmi e complessità computazionale

Problema, istanza di un problema, soluzione ammissibile

- Un **problema astratto** viene definito descrivendo i dati del problema, le variabili del problema stesso e l'obiettivo che si vuole raggiungere con la soluzione del problema, operando sui possibili valori delle sue variabili
  - Es.: dato un insieme di n numeri voglio trovare il valore massimo (o il minimo o il valore medio); dato un grafo G e due vertici $u, v \in V(G)$ voglio trovare, se esiste, il cammino più breve da u a v

- **Istanza di un problema**: l'insieme delle informazioni necessarie per definire i termini di un **problema concreto** ed attuare un procedimento di risoluzione
  - Es.: l'insieme degli n numeri di cui voglio trovare il massimo (o il minimo o la media); le liste di adiacenza o la matrice di adiacenza del grafo G e i due vertici u e v tra i quali voglio trovare il cammino più breve (se esiste)

- **Insieme delle soluzioni ammissibili (spazio delle soluzioni)**: l'insieme di tutte le possibili configurazioni delle variabili di un'istanza di un problema, che soddisfano i requisiti e i vincoli imposti dal problema stesso e dalla specifica istanza che si sta considerando
  - Es.: nel problema del calcolo del massimo in un insieme di n numeri, l'insieme delle soluzioni ammissibili è composto da tutti gli elementi dell'insieme; nel problema della ricerca del cammino più breve tra due vertici di un grafo, l'insieme delle soluzioni ammissibili è costituito da tutti i cammini esistenti tra i due vertici presi in considerazione

Algoritmi

- Un **algoritmo** è una formalizzazione di un procedimento di calcolo
- In matematica, prima ancora che in informatica, il concetto di algoritmo si trova ovunque: ogni espressione aritmetica rappresenta un procedimento di calcolo e dunque un algoritmo che si deve eseguire per calcolare il valore di una funzione in un punto o il valore numerico di un'espressione aritmetica, una volta definito il valore delle singole variabili
  - Es.: una sommatoria descrive in modo sintetico l'operazione di somma iterata più volte, al variare di un indice in un determinato insieme numerico

$$\sum_{i=1}^{n}\left(\sum_{j=i}^{k} x_i t_j\right)$$

- **Algoritmo**: procedura che rispetta i seguenti requisiti:
  1. è costituita da passi elementari (alla portata delle «competenze» e delle «capacità» dell'esecutore);
  2. i passi sono in numero finito;
  3. termina dopo aver eseguito un numero finito di istruzioni (per ogni istanza del problema).
- L'esecuzione di un algoritmo, quindi, termina sempre dopo un tempo finito, individuando la soluzione del problema, ovvero dichiarando che la soluzione non esiste o che non può essere trovata
- NOTA: *il termine «algoritmo» deriva da una latinizzazione del nome del matematico, astronomo, geografo persiano Muḥammad ibn Mūsā al-Khwārizmī, padre dell'algebra vissuto nel 800 d.C.*

Esempio

- È centrale nel concetto di algoritmo la **reiterazione**, la **ripetizione** per un numero anche molto elevato di volte (**purché finito**), di un insieme di operazioni elementari
- Ad esempio consideriamo il seguente algoritmo per il calcolo del minimo comune multiplo tra due interi positivi x e y:

Algoritmo 1 MCM(x, y)

1: siano $m_x := x$ e $m_y := y$
2: se $m_x < m_y$ allora $m_x := m_x + x$ altrimenti se $m_y < m_x$ allora $m_y := m_y + y$
3: se $m_x \neq m_y$ allora vai al passo 2
4: il minimo comune multiplo tra x e y è m_x

- I passi 2 e 3 vengono ripetuti più volte fino a quando m_x e m_y non assumono lo stesso valore; i passi sono in numero finito e sono tutti «elementari» (si tratta di somme e confronti tra interi). Come possiamo essere sicuri che prima o poi termini e non prosegua i calcoli all'infinito senza trovare la soluzione?
  - Dati $x, y > 0$ certamente xy è un multiplo sia di x che y; m_x e m_y sono multipli di x e y rispettivamente, quindi nel caso peggiore ad un certo punto raggiungeranno entrambi il valore xy (oppure l'algoritmo potrà terminare prima con due valori di m_x e m_y minori di xy)
  - Siccome procediamo per incrementi successivi del multiplo minore, il multiplo comune su cui l'algoritmo termina sarà il più piccolo (il minimo multiplo comune ad x e y)

Esempio

m_x	m_y
4	7
8	**7**
8	14
12	14
16	**14**
16	21
20	21
24	**21**
24	28
28	28

m_x	m_y
9	12
18	**12**
18	24
27	**24**
27	36
36	36

Algoritmo 1 MCM(x, y)

1: siano $m_x := x$ e $m_y := y$

2: se $m_x < m_y$ allora $m_x := m_x + x$ altrimenti se $m_y < m_x$ allora $m_y := m_y + y$

3: se $m_x \neq m_y$ allora vai al passo 2

4: il minimo comune multiplo tra x e y è m_x

Esempio (sbagliato)

- Dato un intero $n > 0$ si vuole calcolare la radice quadrata di n

Algoritmo 2 RADICE(n)

1: leggi n
2: $i := 0$
3: $i := i + 1$
4: se $i \times i = n$ allora scrivi i e fermati
5: altrimenti vai al passo 3

- Anche in questo caso la procedura è composta da un numero finito di passi elementari...
- ... **ma purtroppo non funziona**! Ci sono casi (*istanze* del problema) in cui produce il risultato esatto ed altri in cui invece non produce alcun risultato

- Esempio:
  - se $n = 9$, la procedura funziona e trova che la radice quadrata è $i = 3$
  - se invece $n = 8$, la procedura non termina mai la sua esecuzione (8 non è un quadrato perfetto, la radice non è un numero intero!)

Pseudo-codifica di un algoritmo

- Per descrivere un algoritmo utilizziamo uno «pseudo-linguaggio», con cui componiamo lo pseudo-codice di un algoritmo:
  - **leggi**: acquisisce in input un dato e lo memorizza in una variabile (es.: lettura di un dato inserito dall'utente con la tastiera del computer o del terminale)
  - **scrivi**: visualizza in output un dato (es.: visualizzazione sullo schermo del computer o del terminale usato dall'utente)
  - **assegna**: assegna ad una variabile un valore costante, un valore presente in un'altra variabile o risultato di un'operazione tra variabili e valori costanti (es.: $y := 3 + x$)
  - **se … allora … altrimenti …**: valuta una condizione booleana definita come confronto fra variabili o costanti e sceglie se eseguire determinate istruzioni (se la condizione booleana è vera) oppure altre (se la condizione è falsa)
  - **fintanto che … ripeti le seguenti istruzioni …**: esegue le istruzioni indicate fintanto che la condizione booleana indicata è vera (termina l'esecuzione delle istruzioni, quando la condizione diventa falsa)
  - **stop**: termina l'esecuzione dell'algoritmo

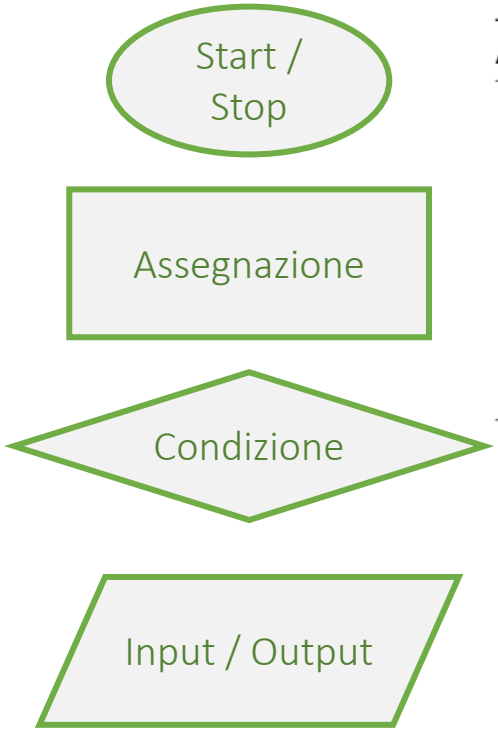
Programmazione strutturata

- È opportuno descrivere gli algoritmi in pseudo-codice, perché è più generale, non ci vincola ad un linguaggio di programmazione specifico e alleggerisce la notazione
- Utilizzeremo un linguaggio **imperativo/procedurale**: è centrale il concetto di **variabile**, l'algoritmo esegue continue assegnazioni e riassegnazioni di valori alle variabili fino ad arrivare ad una condizione di *stop* in cui termina l'algoritmo
- Adotteremo le regole della **programmazione strutturata**: nella progettazione di un algoritmo possono essere utilizzate solo tre strutture
  - **sequenziale**: le istruzioni si susseguono una dopo l'altra
  - **condizionale**: il flusso dell'algoritmo si biforca sulla base del valore assunto da una condizione formulata mediante un'espressione booleana: nel caso l'istruzione sia vera vengono eseguite determina espressione, mentre se è falsa ne vengono eseguite altre; quindi l'algoritmo si ricongiunge in un punto unico e prosegue con l'istruzione successiva alla struttura condizionale
  - **iterativa**: vengono eseguite (reiterate) più volte determinate istruzioni, fino a quando una condizione booleana che regola l'esecuzione delle istruzioni presenti nella struttura iterativa non diventa falsa

  Le tre strutture possono essere **concatenate** fra loro o **nidificate** (completamente contenuta una nell'altra), ma **non possono accavallarsi** (intersecarsi senza la completa inclusione di una struttura in un'altra)

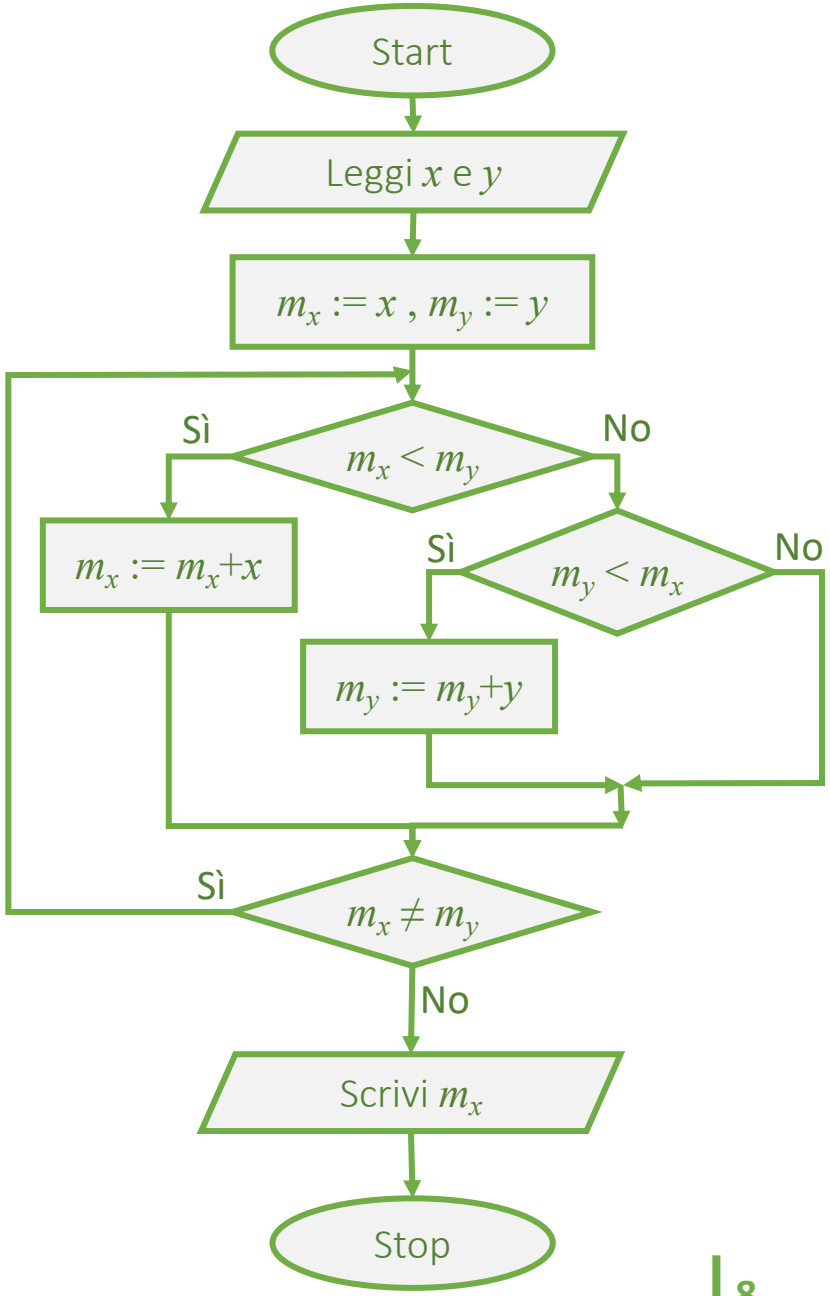
Diagrammi di flusso

- Un algoritmo può anche essere rappresentato con un **diagramma di flusso**, con cui è più semplice studiarne la struttura
- Per costruire un diagramma di flusso si associa ad ognuna delle istruzioni precedenti un simbolo grafico e si collegano tra di loro le istruzioni con delle frecce ad indicare la sequenza delle istruzioni (quale istruzione deve essere eseguita dopo aver concluso l'esecuzione della precedente)



Algoritmo 1 MCM(x, y)

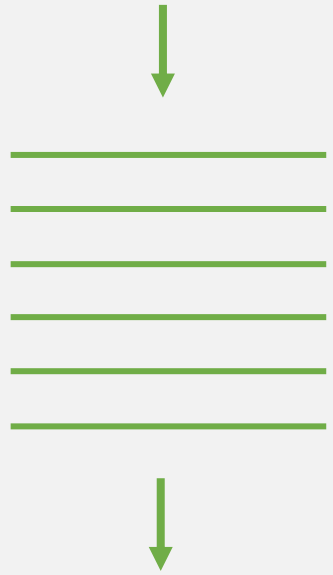
1: siano $m_x := x$ e $m_y := y$
2: se $m_x < m_y$ allora $m_x := m_x + x$ altrimenti se $m_y < m_x$ allora $m_y := m_y + y$
3: se $m_x \neq m_y$ allora vai al passo 2
4: il minimo comune multiplo tra x e y è m_x



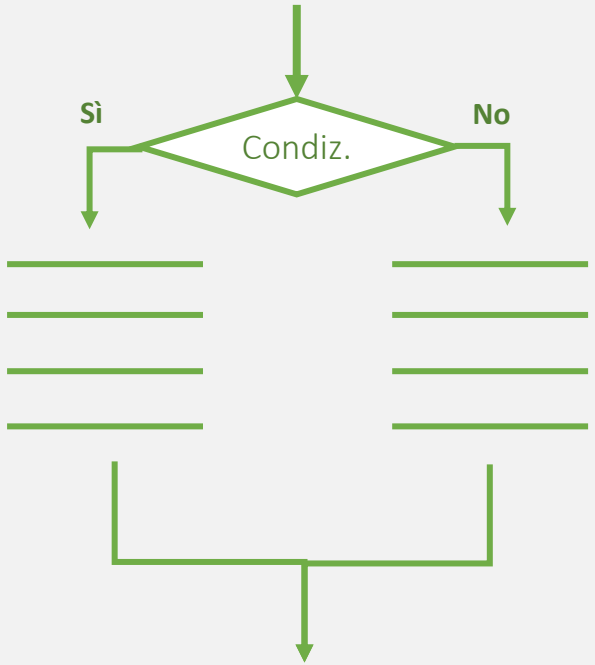
Diagrammi di flusso e programmazione strutturata

- Le tre strutture della **programmazione strutturata** possono essere rappresentate come segue utilizzando un diagramma di flusso

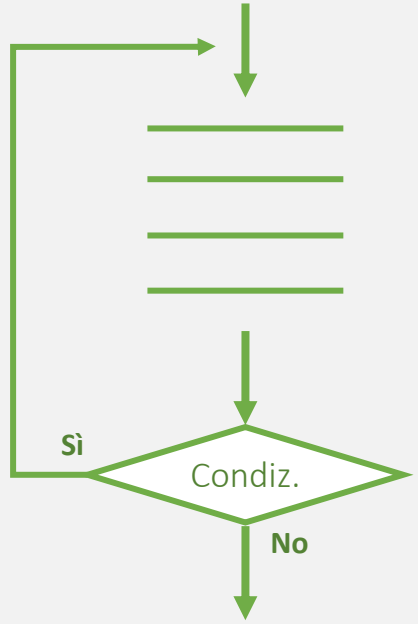
Sequenziale



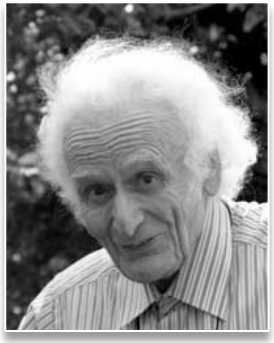
Condizionale



Iterativa



Teorema Fondamentale della Programmazione Strutturata



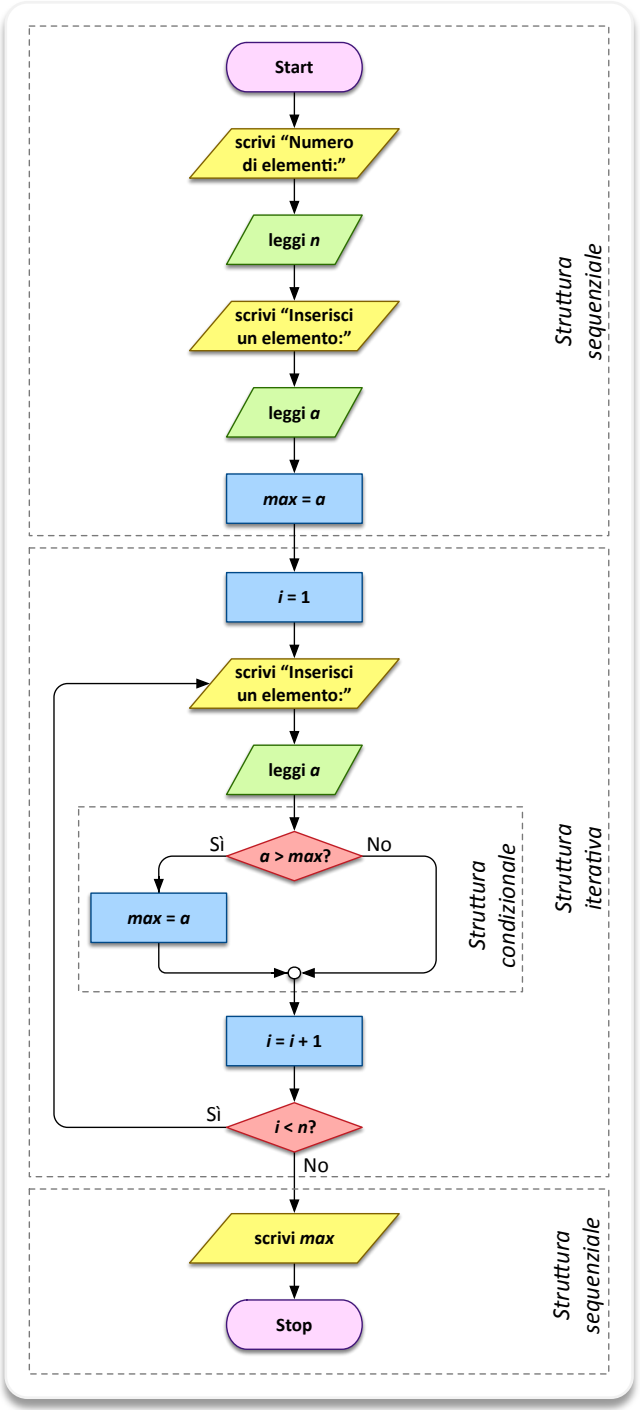
Corrado Böhm
(1923 – 2017)

- Nel 1966 i matematici italiani **Corrado Böhm** e **Giuseppe Jacopini** dimostrarono un teorema tra i più importanti nell'ambito dell'informatica teorica:il **teorema fondamentale della programmazione strutturata** che porta il loro nome

- Il Teorema afferma che qualunque algoritmo risolutore per un problema può essere riscritto applicando le regole della programmazione strutturata

  In altri termini il teorema afferma che la programmazione strutturata **non introduce nessun limite** alla possibilità di codificare un algoritmo per risolvere un problema dato: se il problema è risolubile per via algoritmica, allora è possibile codificare tale algoritmo attenendosi alle regole della programmazione strutturata

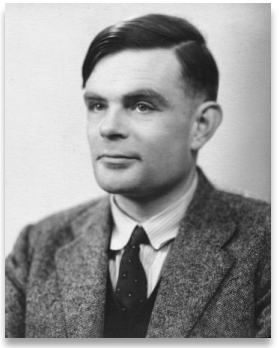
- Tutti i moderni linguaggi di programmazione imperativi/procedurali adottano il paradigma della programmazione strutturata e ne semplificano l'utilizzo da parte del programmatore



Algoritmi e calcolabilità

- L'algoritmo (sbagliato) per il calcolo della radice quadrata mette in evidenza un fatto cruciale: gli algoritmi, per essere corretti, devono risolvere **ogni istanza** del problema e non solo alcune
- Esistono problemi la cui soluzione non è calcolabile mediante un algoritmo per qualche istanza?
- La risposta è stata data da Alan Turing (1912–1954).
  - **Alan Turing** propone un modello di calcolo, la «**macchina di Turing**», che definisce in modo più rigoroso ed essenziale il concetto di algoritmo.
  - **Alonzo Church** (1903-1995), logico-matematico, ha insegnato praticamente sempre all'università di Princeton. È l'autore del «**lambda calcolo**», da cui poi hanno tratto ispirazione i linguaggi funzionali come il Lisp.
  - **Tesi di Church–Turing**: *se un problema è calcolabile, allora esiste una macchina di Turing (o un dispositivo equivalente, come il computer) in grado di risolverlo (cioè di calcolarlo). La classe delle funzioni calcolabili coincide con quella delle funzioni calcolabili da una macchina di Turing.*
- Church e Turing dimostrano (indipendentemente l'uno dall'altro) l'esistenza di un problema indecidibile, ossia un problema non calcolabile con una Macchina di Turing.
  - **Problema della fermata** (Turing 1937): data una Macchina di Turing M qualsiasi e un'istanza I di un problema, M si fermerà dopo un numero finito di passi elaborando I?
  - Si può dimostrare che non è possibile definire una macchina di Turing (un algoritmo) per risolvere il problema della fermata.
- Quindi **esistono problemi non calcolabili**: il «problema della fermata» è uno di questi

Alonzo Church
(1903 – 1995)



Alan Turing
(1912 – 1954)

Algoritmi e complessità computazionale

- La **complessità computazionale** di un algoritmo è una **funzione** che esprime il **numero di operazioni elementari** che devono essere eseguite per calcolare la soluzione di un'istanza di un problema, **in funzione della dimensione dell'istanza** stessa

- La **dimensione dell'istanza di un problema** è data dal numero di informazioni necessarie per definirla (es.: il numero di elementi di un insieme su cui bisogna eseguire un calcolo, la dimensione di un grafo, ecc.)

- Dato un algoritmo A che risolve il problema P, la complessità computazionale di A è la funzione

$$f_A : \mathbb{N} \to \mathbb{N}$$

definita associando alla dimensione di un'istanza I_P del problema P, che indichiamo con $|I_P|$, il numero di operazioni compiute da A per risolvere I_P

- Si dice quindi che l'algoritmo A risolve l'istanza I_P del problema P in **tempo** $t = f(|I_P|)$, perché si assume che l'esecuzione di una singola operazione elementare richieda una singola unità di tempo

Algoritmi e complessità computazionale

- La funzione di complessità di un algoritmo è dunque una **misura della sua efficienza** e può quindi essere usata per confrontare l'efficienza di due algoritmi differenti che risolvono lo stesso problema

- A fronte di **due istanze distinte di uno stesso problema**, I_P' e I_P'', di uguale dimensione, uno stesso algoritmo A potrebbe avere **comportamenti differenti**, rendendo non univoco il valore della funzione complessità f_A

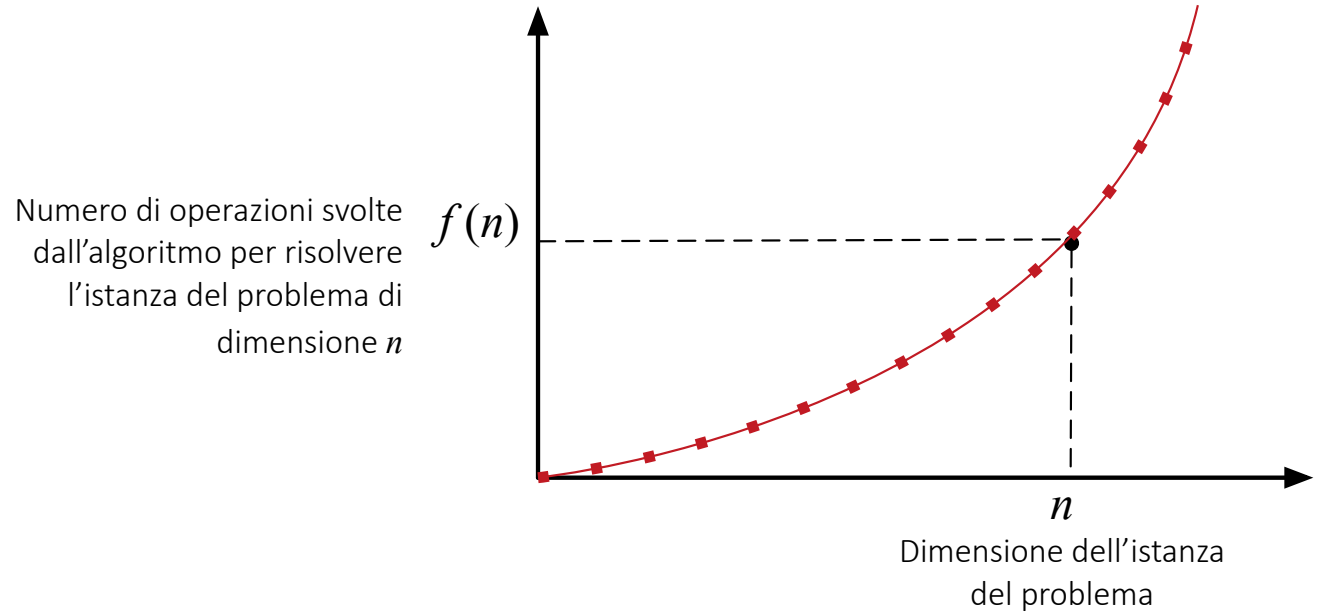
- Per questo motivo si assume come valore della funzione di complessità per una determinata dimensione delle istanze del problema, il tempo (il numero di operazioni) più alto, fra tutte le istanze di uguale dimensione

- In altri termini: la complessità dell'algoritmo è data dal numero di operazioni che si dovrebbero eseguire nel caso peggiore, applicando l'algoritmo all'**istanza del problema meno favorevole** (per l'algoritmo)

Caratteristiche della funzione complessità

La funzione complessità di un algoritmo è

- **definita su** $\mathbb{N}$: il dominio della funzione è la dimensione di un'istanza di un problema (il numero di informazioni che definiscono l'istanza), ossia il numero di dati che devono essere forniti in input all'algoritmo per poter risolvere il problema
- **a valori interi positivi**: la funzione esprime il numero di operazioni eseguite dall'algoritmo; tale numero non può quindi essere negativo, né può essere non intero, per definizione
- in generale sono **monotone crescenti** (o non decrescenti): con l'aumentare della dimensione dell'istanza, aumenta anche il valore della funzione (il numero di operazioni da eseguire per risolvere l'istanza del problema)



Classi di complessità

- Visto che la complessità di un algoritmo esprime la sua efficienza nel caso peggiore (negli altri casi sappiamo solo che **non** impiegherà un tempo maggiore), non è interessante valutarne esattamente il valore in ogni punto, quanto piuttosto **studiare l'andamento «asintotico» della funzione**, al crescere della dimensione delle istanze del problema di cui si vogliono calcolare le soluzioni
- Ciò che interessa è l'andamento asintotico della funzione, si eliminano quindi i termini costanti o i coefficienti costanti (indipendenti dalla dimensione dell'istanza del problema)
  - se $f_A(n) = k_1\varphi(n) + k_2$ la funzione che si andrà a studiare sarà semplicemente $\tilde{f}_A(n) = \varphi(n)$
  - se f_A è un polinomio, se ne considera solo il termine di grado massimo (quello che indirizza l'andamento asintotico dell'intera funzione)
- Così facendo si finisce per aggregare in uno stesso insieme funzioni di complessità differenti: infatti la funzione $f_A(n) = k_1\varphi(n) + k_2$ e $f_B(n) = h_1\varphi(n) + h_2$, eliminando i fattori e i termini costanti diventeranno $\tilde{f}_A(n) = \tilde{f}_B(n) = \varphi(n)$

Classi di complessità

- Si definiscono quindi **tre notazioni** distinte per definire gli insiemi (**classi**) di funzioni in cui aggregare le funzioni di complessità con analogo andamento asintotico:

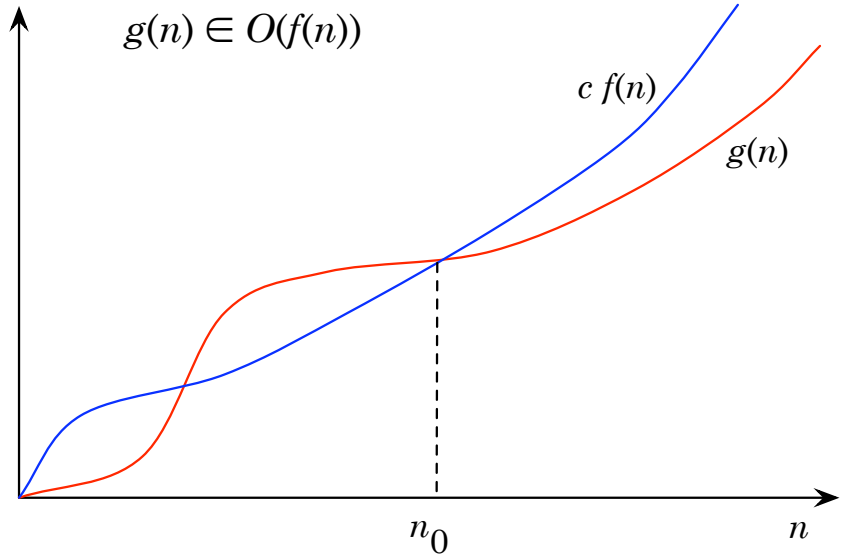
$$O(f(n)) = \{g(n) : \exists\, n_0 > 0, c > 0 \text{ t.c. } g(n) \leq cf(n)\ \forall\, n > n_0\}$$

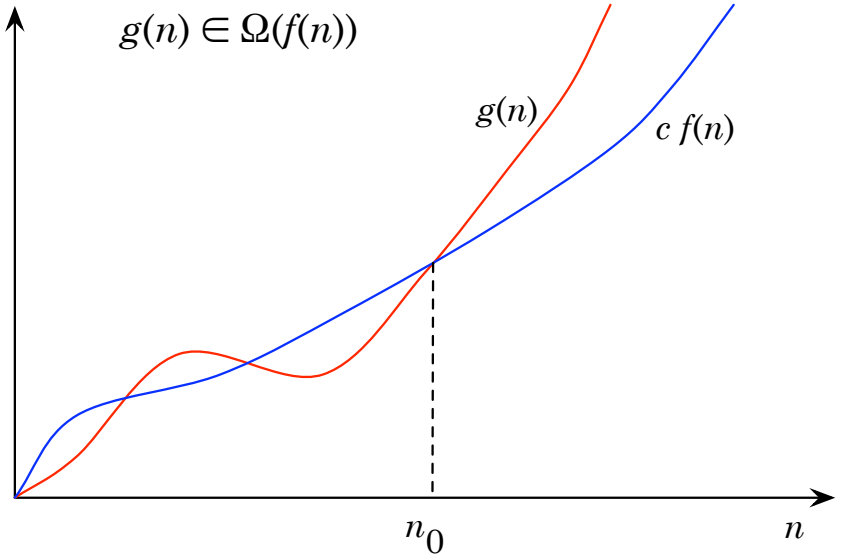
$$\Omega(f(n)) = \{g(n) : \exists\, n_0 > 0, c > 0 \text{ t.c. } g(n) \geq cf(n)\ \forall\, n > n_0\}$$

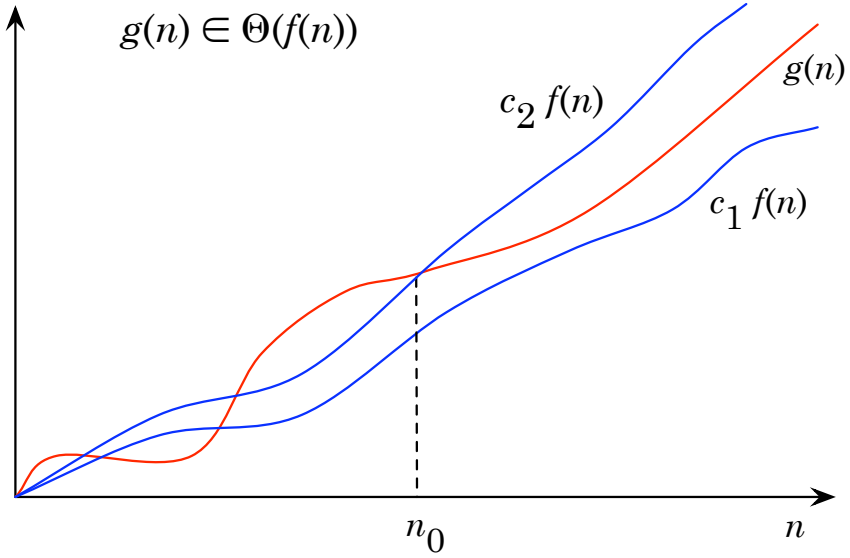
$$\Theta(f(n)) = \{g(n) : \exists\, n_0 > 0, c_1, c_2 > 0 \text{ t.c. } c_1 f(n) \leq g(n) \leq c_2 f(n)\ \forall\, n > n_0\}$$

- In generale quindi non si dirà che l'algoritmo A ha complessità $f(n)$, ma che la sua complessità appartiene alla classe (o è dell'ordine) $O(f(n))$

Classi di complessità







Classi di complessità

Proposizione: Date due funzioni di complessità $f(n)$ e $g(n)$ risulta:

1. $g(n) \in O(f(n)) \iff f(n) \in \Omega(g(n))$
2. $g(n) \in \Theta(f(n)) \iff g(n) \in O(f(n))$ **e** $g(n) \in \Omega(f(n))$
3. $g(n) \in \Theta(f(n)) \iff f(n) \in \Theta(g(n))$

Dimostrazione:

1. Banale per la simmetria delle definizioni O e Ω
2. Ovvio per le definizioni: basta applicarle e poi scegliere il valore di n_0 più grande
3. $g \in \Theta(f) \implies$ per (2) $g \in O(f)$ e $g \in \Omega(f) \implies$ per la (1) $f \in \Omega(g)$ e $f \in O(g) \implies$ per la (2) $f \in \Theta(g)$

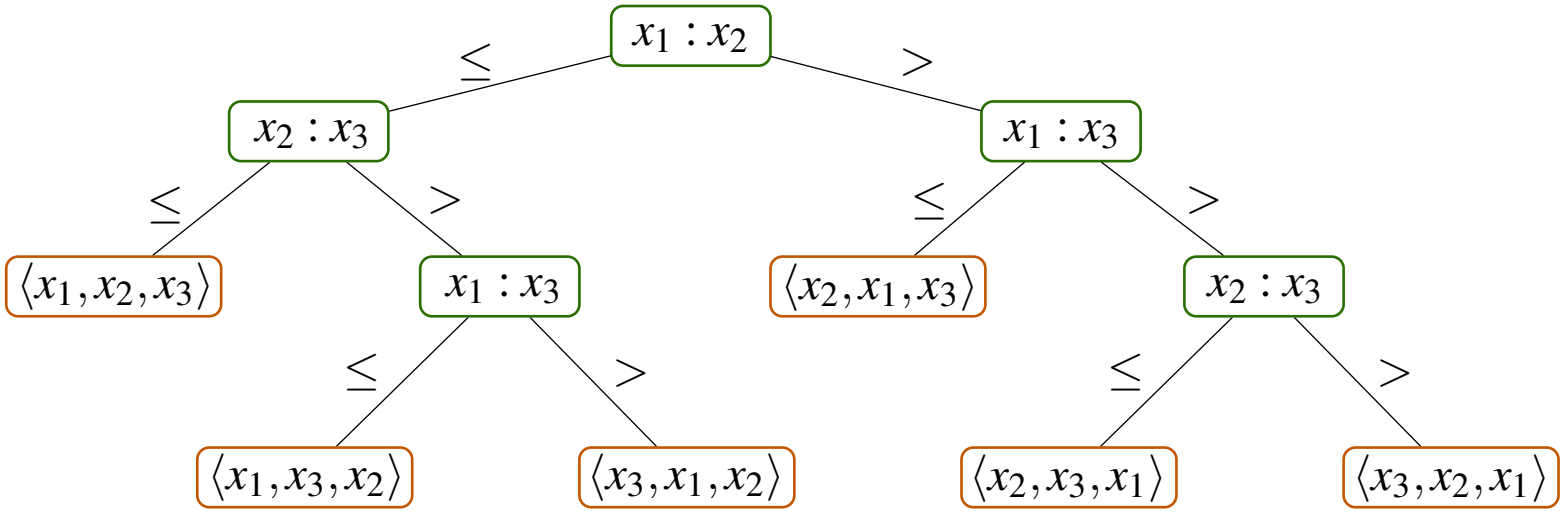
Complessità dei problemi

- La **complessità computazionale di un problema** è data dalla complessità **più bassa** che può avere un algoritmo risolutivo per tale problema
- Un algoritmo che risolve un determinato problema e la cui complessità computazionale coincide con quella del problema, si dice **ottimo**
- **Teorema.** La complessità di un algoritmo che risolve il problema SORT attraverso confronti è $\Omega(n \log_2 n)$

  Dimostrazione. Possiamo rappresentare con un *albero binario* tutte le possibili operazioni di confronto tra gli n elementi della sequenza, effettuati da un qualsiasi algoritmo di ordinamento: ad ogni nodo c'è il confronto tra due elementi e a seconda dell'esito del confronto se ne eseguirà un altro o un altro ancora.

  Il numero di stati finali è **maggiore o uguale** al numero di permutazioni degli elementi dell'insieme da ordinare: $n!$

  La massima profondità dell'albero di decisione è pari al massimo numero di confronti eseguiti per risolvere il problema di ordinamento



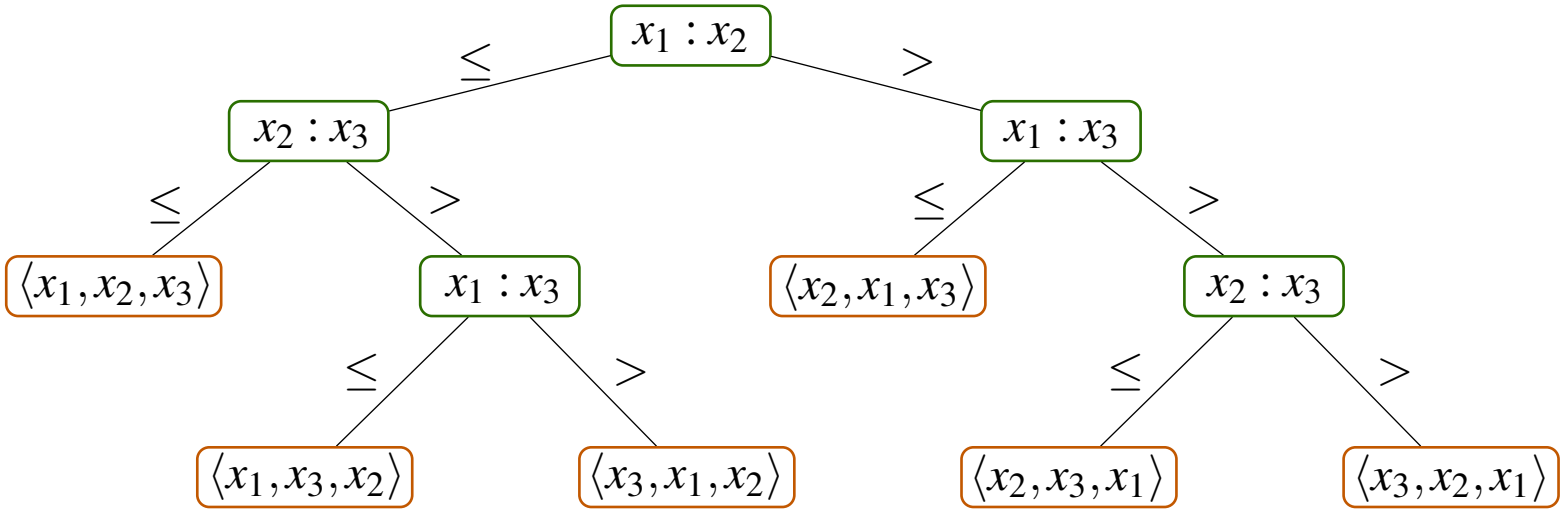
Complessità dei problemi

- **Teorema.** La complessità di un algoritmo che risolve il problema SORT attraverso confronti è $\Omega(n \log_2 n)$

  Dimostrazione. Possiamo rappresentare con un *albero binario* tutte le possibili operazioni di confronto tra gli n elementi della sequenza, effettuati da un qualsiasi algoritmo di ordinamento: ad ogni nodo c'è il confronto tra due elementi e a seconda dell'esito del confronto se ne eseguirà un altro o un altro ancora.

  Il numero di stati finali è **maggiore o uguale** al numero di permutazioni degli elementi dell'insieme da ordinare: $n!$

  La massima profondità dell'albero di decisione è pari al massimo numero di confronti eseguiti per risolvere il problema di ordinamento

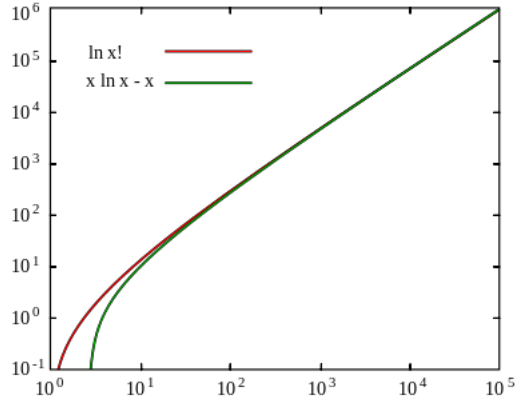


Supponiamo che l'albero abbia profondità h allora avrà al massimo 2^h foglie (i vertici dell'albero binario ad ogni livello al massimo raddoppiano rispetto al livello precedente)

Quindi $2^h \geq n!$ per cui $h \geq \log_2 n!$

Utilizzando la formula di Stirling per approssimare il fattoriale, si ottiene $h \geq n \log_2 n$

Quindi il numero di operazioni svolte da qualsiasi algoritmo risolutore è maggiore o uguale a $n \log_2 n$ per cui la complessità del problema è $\Omega(n \log_2 n)$ ■



Complessità dei problemi e algoritmi ottimi

- Dal Teorema precedente si ricava che gli algoritmi HEAPSORT e MERGESORT, che hanno complessità $O(n \log_2 n)$ sono **algoritmi ottimi**
- Infatti, sia $g(n)$ la funzione di complessità dei due algoritmi: $g(n) \in O(n \log_2 n)$
- Allora: $g(n) \leq cn \log_2 n$ per qualche $c > 0$ e per $n > n_0$

  per il Teorema però $g(n) \geq n \log_2 n$

  Quindi la complessità degli algoritmi coincide con quella del problema, e dunque i due algoritmi sono ottimi

Problemi polinomiali (trattabili)

- Indichiamo con P l'insieme dei **problemi di complessità polinomiale** (risolubili da un algoritmo di complessità polinomiale), del tipo $O(n^k)$ per qualche $k > 0$
- Naturalmente, applicando la definizione, risulta $O(n^k) \subset \mathrm{O}(n^{k+1})$
- $O(n \log_2 n) \in P$ infatti $O(n \log_2 n) \subset O(n^2) \in P$

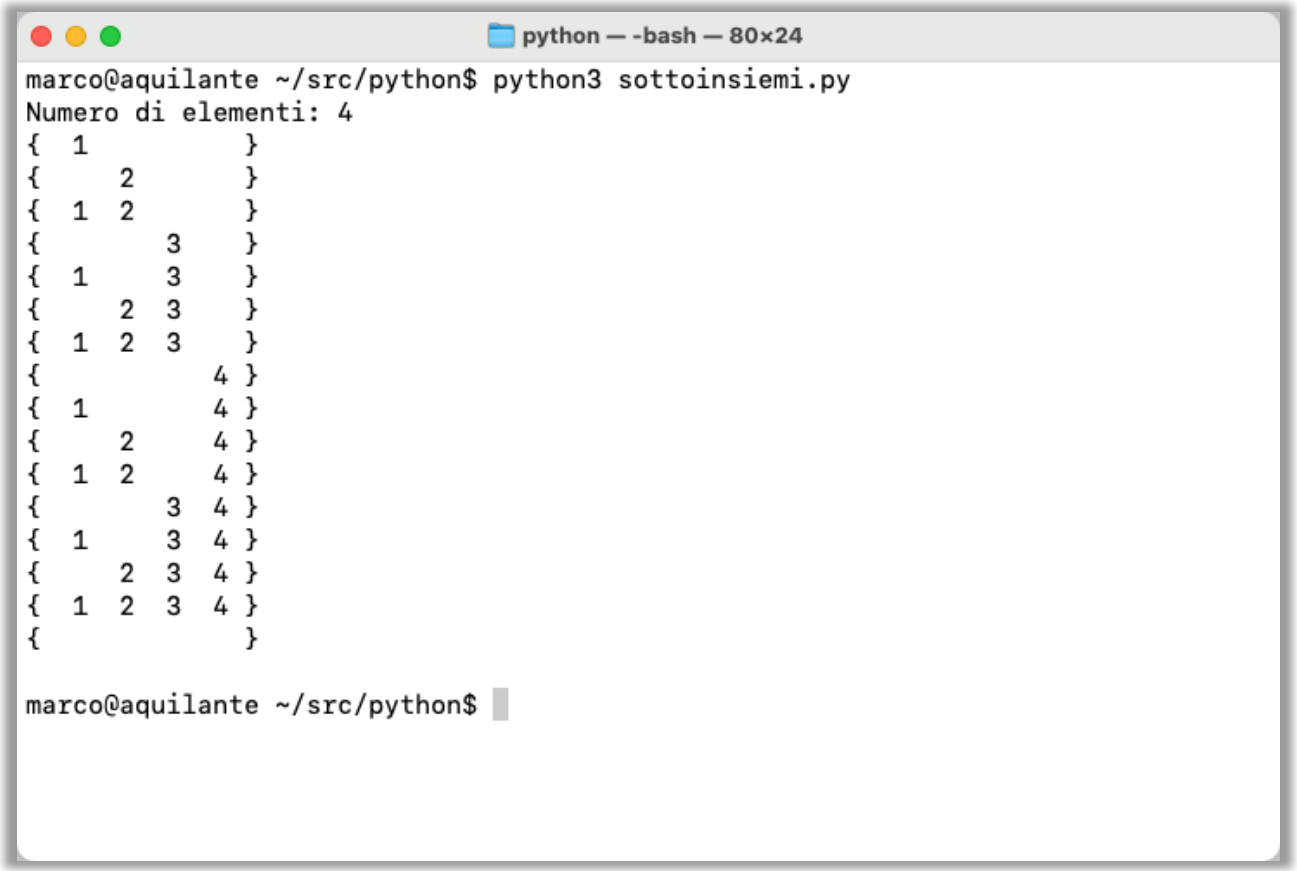
- I problemi di complessità polinomiale sono **trattabili**: l'algoritmo risolutore calcola la soluzione in un tempo «ragionevole». anche per grandi valori di n

- Se il nostro computer esegue **un milione di istruzioni al secondo**, per risolvere un problema di complessità $O(n^3)$, con $n = 100$, eseguirà approssimativamente *un milione di istruzioni* impiegando circa 1 secondo di tempo
- Se l'istanza del problema cresce di 10 volte ($n = 1.000$) dovranno essere eseguite $1.000.000.000$ di operazioni in un tempo di circa 17 minuti

Problemi super-polinomiali

- Non tutti i problemi sono di complessità polinomiale
- Ad esempio se un problema richiede la valutazione di tutti i sottoinsiemi di un set di n dati, dovrà elaborare 2^n sottoinsiemi

- Facciamo una prova...



Problemi super-polinomiali (intrattabili)

- I problemi di complessità **super-polinomiale** ($O(k^n)$, $O(n!)$, ...) sono **intrattabili**: il tempo per il calcolo della soluzione è irragionevole (troppo elevato) anche per piccoli valori di n
- Supponiamo che il problema A abbia complessità $O(2^n)$; supponiamo che il nostro computer sia in grado di eseguire un milione di operazioni al secondo.
- Se $n = 50$ allora l'algoritmo dovrà eseguire approssimativamente $2^n = 1.125.899.906.842.620$ operazioni per risolvere il problema (*un milione di miliardi di operazioni*) impiegando più di un miliardo di secondi per calcolare la soluzione: **più di trent'anni ininterrotti di calcolo**...
- Quel che è peggio, però, è che se aumentiamo di un solo elemento la dimensione dell'istanza del problema ($n = 51$) il tempo di calcolo della soluzione... raddoppia: più di 60 anni!

*allora scegliamo un computer più potente: da **100 milioni di operazioni al secondo**...!*

	Dimensione del problema					
$f(n)$	$n = 10$	$n = 20$	$n = 40$	$n = 50$	$n = 100$	$n = 1.000$
$\log_2 n$	$3 \cdot 10^{-8}$ sec.	$4 \cdot 10^{-8}$ sec.	$5 \cdot 10^{-8}$ sec.	$5 \cdot 10^{-8}$ sec.	$6 \cdot 10^{-8}$ sec.	10^{-7} sec.
n	10^{-7} sec.	$2 \cdot 10^{-7}$ sec.	$4 \cdot 10^{-7}$ sec.	$5 \cdot 10^{-7}$ sec.	10^{-6} sec.	10^{-5} sec.
$n \log_2 n$	$3 \cdot 10^{-7}$ sec.	$8 \cdot 10^{-7}$ sec.	$2 \cdot 10^{-6}$ sec.	$2 \cdot 10^{-6}$ sec.	$6 \cdot 10^{-6}$ sec.	0,0001 sec.
n^2	10^{-6} sec.	$4 \cdot 10^{-6}$ sec.	0,000016 sec.	0,000025 sec.	0,0001 sec.	0,01 sec.
n^3	0,00001 sec.	0,00008 sec.	0,00064 sec.	0,00125 sec.	0,01 sec.	10 sec.
n^4	0,0001 sec.	0,0016 sec.	0,0256 sec.	0,0625 sec.	1 sec.	16 min.
2^n	0,00001 sec.	0,01 sec.	3 ore	130 gg	miliardi di anni	...
$n!$	0,036 sec.	7 secoli	miliardi di anni	...	...	...

NP: Nondeterministic Polynomial time

- Indichiamo con NP l'insieme dei **problemi che possono essere risolti in tempo polinomiale** da un algoritmo/macchina «**non deterministica**»
- Una macchina non deterministica è un modello di calcolo in grado di utilizzare contemporaneamente un numero arbitrario di esecutori/processori con un'istruzione del tipo

  ```
  x := choice(A)
  ```

  che dice di procedere con le istruzioni successive con tanti processi paralleli quanti sono i possibili valori di **A** che possono essere assegnati ad **x**
- Ogni processore esegue un algoritmo che termina dopo un numero finito di passi, individuando la soluzione o arrivando ad un punto in cui è evidente che la configurazione individuata non è una soluzione del problema
- Quando un algoritmo termina lo comunica a tutti gli altri, indicando se ha raggiunto la soluzione del problema (`success`) o se ha terminato l'esecuzione senza trovarla (`failure`)
- Se un processo termina nello stato *success*, vengono interrotti anche tutti gli altri

Esempi di algoritmi non deterministici

- Ordinare gli n elementi dell'insieme A

Algoritmo 4 ORDINAMENTO$(A = \langle a_1, \ldots, a_n \rangle)$

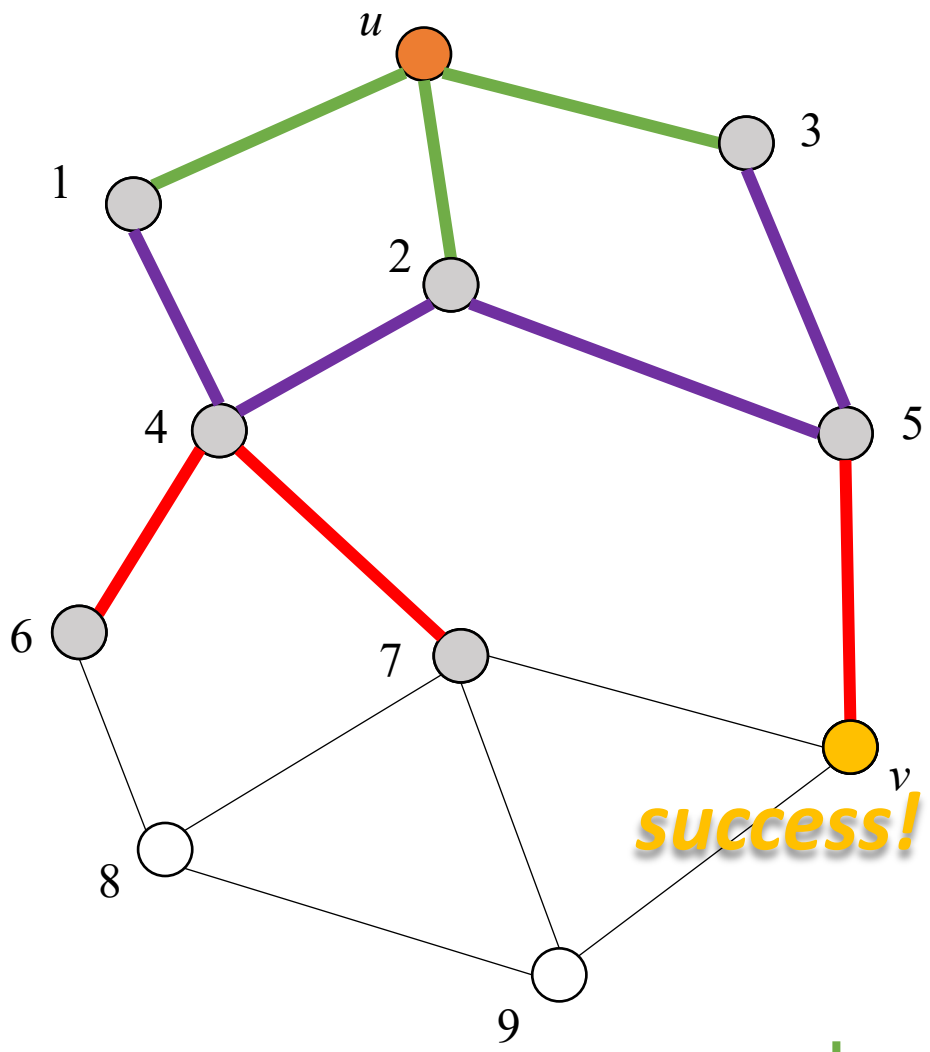
```
1:  i := choice(1, ..., n)
2:  scambia(a_1, a_i)
3:  per i := 2, ..., n ripeti
4:      j := choice(i, ..., n)
5:      scambia(a_i, a_j)
6:      se a_i < a_{i-1} allora
7:          failure
8:      fine-condizione
9:  fine-ciclo
10: success
```

Esempi di algoritmi non deterministici

- Individuare il cammino con il minimo numero di spigoli (cammino di lunghezza minima) tra due vertici u e v su un grafo G

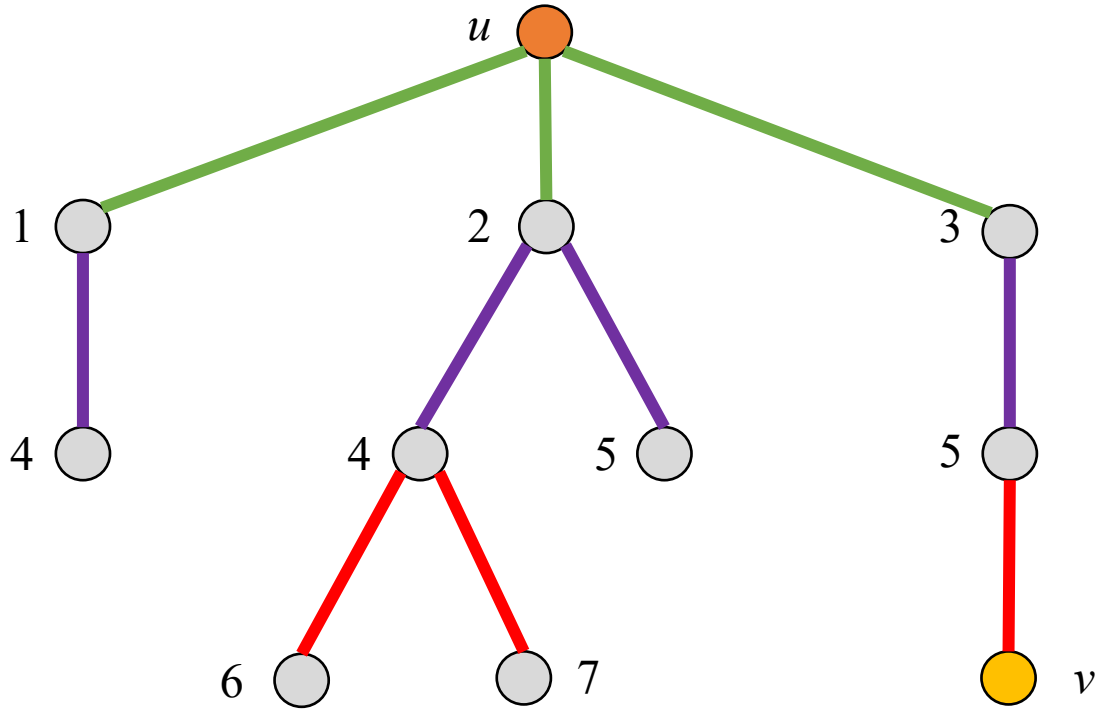
Algoritmo 3 CAMMINOMINIMO(G,u,v)

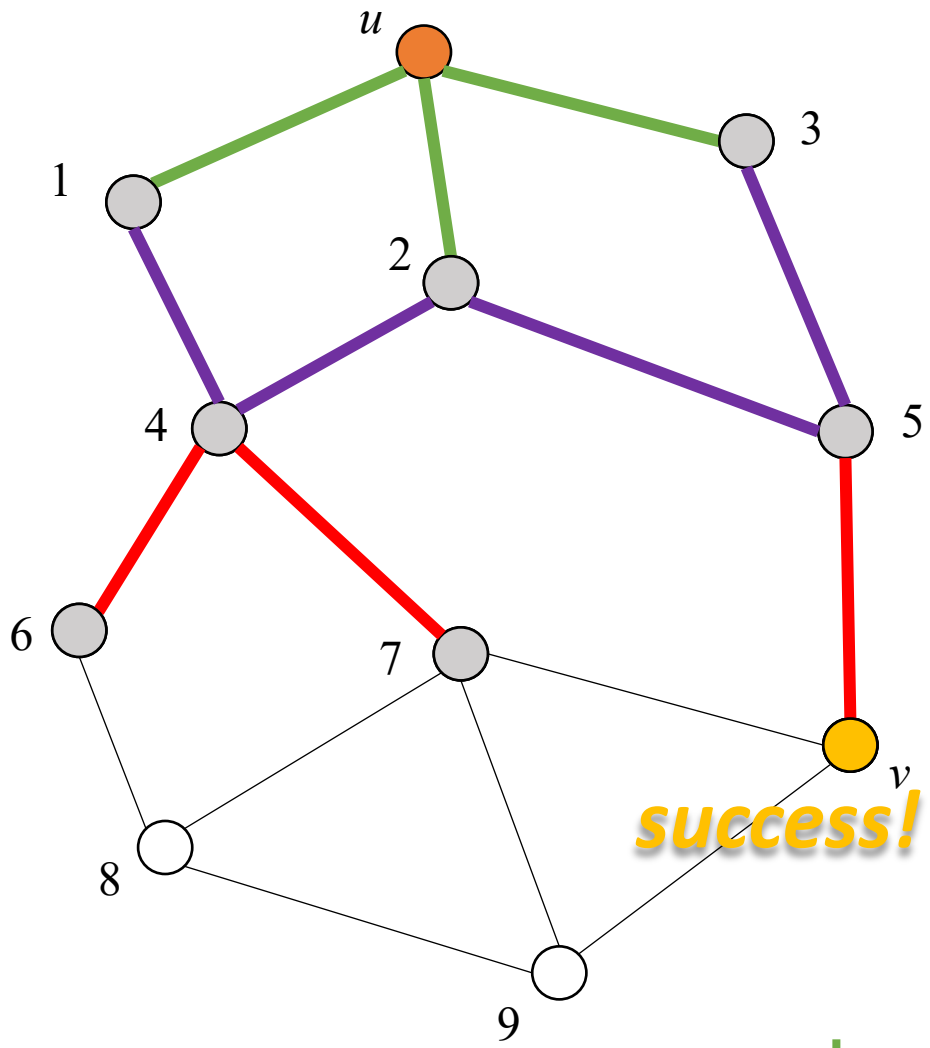
1: $x :=$ choice$(N(u))$
2: **se** $x = v$ **allora**
3: *success*
4: **altrimenti**
5: **se** colore(x) = grigio oppure $x = u \implies$ *failure*
6: **altrimenti** colore$(x) :=$ grigio
CAMMINOMINIMO(G,x,v)
7: **fine-condizione**



Esempi di algoritmi non deterministici

- Individuare il cammino con il minimo numero di spigoli (cammino di lunghezza minima) tra due vertici u e v su un grafo G





NP: Nondeterministic Polynomial time

- I **problemi** NP sono quelli che sono risolubili in tempo polinomiale da una macchina/algoritmo non deterministica
- L'esecuzione di un algoritmo di questo genere può essere rappresentata come un albero (non necessariamente binario), in cui le «ramificazioni» si sviluppano proprio in corrispondenza delle istruzioni `choice`
- Di fatto, se la soluzione esiste, allora può essere calcolata con le operazioni identificate dai vertici dell'albero posti sul cammino dalla radice dell'albero fino alla foglia che identifica tale soluzione
- Il numero di operazioni h è tale che esiste una costante $k > 0$ per cui risulti $h \leq n^k$, se n è la dimensione dell'istanza del problema
- Se si vuole controllare che una determinata sequenza di scelte consenta di determinare una soluzione del problema, basterà ripercorrere il «cammino» (sull'albero delle decisioni) corrispondente a tali scelte
- Quindi una definizione alternativa della classe dei problemi NP è la seguente: *l'insieme di tutti i* ***problemi le cui soluzioni possono essere verificate in tempo polinomiale*** *da un algoritmo deterministico (ammettono un* ***algoritmo di verifica*** *di complessità polinomiale)*
- Naturalmente $P \subseteq NP$

Riducibilità di un'istanza di un problema ad un'istanza di un altro problema

- Dati due problemi A e B, è possibile **ridurre** A a B se esiste un algoritmo R: $I_A \longrightarrow I_B$ con cui è possibile trasformare ogni istanza I_A del problema A in una istanza equivalente I_B del problema B, in modo tale che la soluzione di I_B sia anche una soluzione di I_A

- Se l'algoritmo R è di complessità polinomiale $O(n^k)$ allora diremo che A è **riducibile in tempo polinomiale** a B e scriveremo $A \leq_P B$

- Se $A \leq_P B$ e se $B \in P$ allora anche $A \in P$

- Se $A \leq_P B$ allora per risolvere A posso tradurre l'istanza di A in un'istanza equivalente di B, poi risolvere B e così ottenere anche una soluzione per A; dunque A **non è più difficile** di B: se riesco a risolvere B in tempo polinomiale, allora posso risolvere anche A in tempo polinomiale

- La riducibilità in tempo polinomiale è **transitiva**: se $A \leq_P B$ e $B \leq_P C$ allora $A \leq_P C$ perché la «concatenazione» di algoritmi polinomiali, dà luogo ad un algoritmo di complessità polinomiale

Problemi NP-completi

- Un problema A è **NP-completo** (NPC) se
  - $A \in NP$
  - per ogni $B \in NP$ risulta $B \leq_P A$

- **SAT**: dato un insieme U di variabili booleane (che possono assumere valori in $\{vero, falso\}$) e un insieme di clausole su U, esiste una assegnazione di valori alle variabili di U tale da rendere vera l'espressione?
  - Esempio: $((x_1 \wedge x_2) \vee (x_1 \wedge x_3)) \wedge (x_2 \iff \neg x_3)$

    L'espressione risulta vera per $x_1 = vero$, $x_2 = vero$ e $x_3 = falso$, ma anche per $x_1 = vero$, $x_2 = falso$ e $x_3 = vero$, mentre per ogni altra assegnazione di valori l'espressione è falsa
  - Esempio: $(x_1 \wedge x_2) \wedge (x_1 \wedge \neg x2)$

    L'espressione non ammette alcuna assegnazione di valori alle variabili x_1 e x_2 tale da renderla vera

Problemi NP-completi


Stephen A. Cook
(1939)


Leonid A. Levin
(1948)

- **Teorema di Stephen Cook** (e Leonid Levin) – 1971: il problema SAT è NP-completo

  Dimostrazione. La dimostrazione che SAT è NP è banale: per verificare una soluzione, basta assegnare i valori alle variabili e calcolare la formula

  Che SAT sia NP-Completo lo si dimostra basandosi sul fatto che ogni algoritmo o macchina di Turing può essere assimilato ad una espressione booleana e dunque ogni istanza di un problema NP può essere codificata in tempo polinomiale in una formula booleana equivalente.

- A partire dal Teorema di Cook, per dimostrare che il problema A è NP-completo basta mostrare che esiste un problema NP-completo B che sia riducibile in tempo polinomiale ad A
- Schema di dimostrazione che $A \in NPC$:
  1. si dimostra che $A \in NP$ (provando che esiste un algoritmo di verifica della soluzione, di complessità polinomiale)
  2. Si sceglie un opportuno $B \in NPC$ e si dimostra che $B \leq_P A$
- Infatti, se $B \in NPC$, allora per ogni $C \in NP$ risulta $C \leq_P B$; quindi se $B \leq_P A$, allora anche $C \leq_P A$ per ogni $C \in NP$

Problemi NP-completi: alcuni esempi

- 3DM (*three-dimensional matching*):dati tre insiemi finiti X, Y e Z, disgiunti e con lo stesso numero k di elementi, possiamo definire l'insieme $T \subseteq X \times Y \times Z$

  T contiene un matching? Ossia, esiste un sottoinsieme $M \subseteq T$ di cardinalità $|M| = k$ tale che gli elementi di M abbiano a due a due nessun elemento in comune? Ossia, per ogni $a, b, \in M$ deve risultare $a = (x_1, y_1, z_1)$, $b = (x_2, y_2, z_2)$ e $x_1 \neq x_2, y_1 \neq y_2$ e $z_1 \neq z_2$

- Esempio: Sia $X = \{1, 2, 3\}$, $Y = \{4, 5, 6\}$ e $Z = \{7, 8, 9\}$. Sia inoltre $T = \{ t_1 = (1, 4, 7), t_2 = (2, 5, 7), t_3 = (2, 6, 8), t_4 = (1, 4, 9), t_5 = (3, 5, 9), t_6 = (2, 4, 7), t_7 = (3, 5, 9)\}$

  L'insieme T contiene un *matching* M con $k = 3$ elementi, ad esempio $M = \{ t_1 = (1, 4, 7), t_3 = (2, 6, 8), t_5 = (3, 5, 9) \}$

- Dimostrare che 3DM è NP-Completo è possibile applicando lo schema di dimostrazione:
  - 3DM è NP: dato un *mathing* M è facile verificare se ha k elementi e se le terne sono tutte distinte (l'algoritmo di verifica è certamente polinomiale)
  - è possibile ridurre in tempo polinomiale tutti i problemi NP a 3DM perché SAT $\leq_P$ 3SAT $\leq_P$ 3DM
- È interessante osservare che invece il problema del matching su due soli insiemi (noto anche come *Stable Marriage Problem*) è polinomiale

Problemi NP-completi: alcuni esempi

- PARTITION: dato un insieme finito $A \subset \mathbb{N}$ esiste $A' \subseteq A$ tale che $\sum_{a \in A'} a = \sum_{a \in A \setminus A'} a$

- Esempio: Sia $A = \{1, 4, 5, 7, 8, 12, 17, 22\}$. Il sottoinsieme $A' = \{4, 5, 7, 22\}$ soddisfa la partizione richiesta dal problema: $4+5+7+22 = 1+8+12+17 = 38$

- Dimostrare che PARTITION è NP-Completo è possibile applicando lo schema di dimostrazione:
  - PARTITION è NP: dato un sottoinsieme A' di A è possibile calcolare la somma di A' e la somma di $A \setminus A'$ in tempo polinomiale e confrontare le due somme per verificare se A' soddisfa la partizione richiesta dal problema
  - Per dimostrare che PARTITION è NP-completo si può mostrare che esiste un algoritmo di riduzione polinomiale da 3DM: 3DM $\leq_P$ PARTITION

Problemi NP-completi: alcuni esempi

- SUBSET-SUM: dato un insieme finito $A \subset \mathbb{N}$ e un intero positivo b, esiste un sottoinsieme $A' \subseteq A$ tale che $\sum_{a \in A'} a = b$?

- Esempio: Sia $A = \{3, 2, 18, 23, 7, 9\}$ e sia $b = 43$; il sottoinsieme $A' = \{2, 18, 23\}$ è una soluzione del problema, infatti $2 + 18 + 23 = 43$; invece, per $b = 40$ il problema non ammette soluzione

- Dimostrare che SUBSET-SUM è NP-Completo è possibile applicando lo schema di dimostrazione:
  - SUBSET-SUM è NP: dato un sottoinsieme A' di A è possibile calcolare la somma di A' in tempo polinomiale e confrontare la somma con l'intero b per verificare se A' soddisfa la richiesta dal problema
  - Per dimostrare che SUBSET-SUM è NP-completo si può mostrare che esiste un algoritmo di riduzione polinomiale da PARTITION : PARTITION $\leq_P$ SUBSET-SUM

Problemi NP-completi: alcuni esempi

- **Proposizione**. PARTITION $\leq_P$ SUBSET-SUM

  Dimostrazione. Vogliamo dimostrare che ogni istanza di PARTITION può essere trasformata in un'istanza di SUBSET-SUM e che la soluzione di questa istanza è anche soluzione dell'istanza di PARTITION ad essa associata. Sia $A \subset \mathbb{N}$ un'istanza di PARTITION. Sia $|A| = n$ e $s = a_1 + a_2 + \ldots + a_n$. L'istanza equivalente di SUBSET-SUM è definita come $(A, s/2)$ ■

- **Proposizione**: SUBSET-SUM $\leq_P$ PARTITION

  Dimostrazione. Consideriamo l'istanza di SUBSET-SUM $(\{a_1, \ldots, a_n\}, k)$. Sia $s = a_1 + a_2 + \ldots + a_n$.

  Possiamo costruire la seguente istanza di PARTITION: $\{a_1, \ldots, a_n, b, c\}$ con $b = 2s - k$ e $c = s + k$.

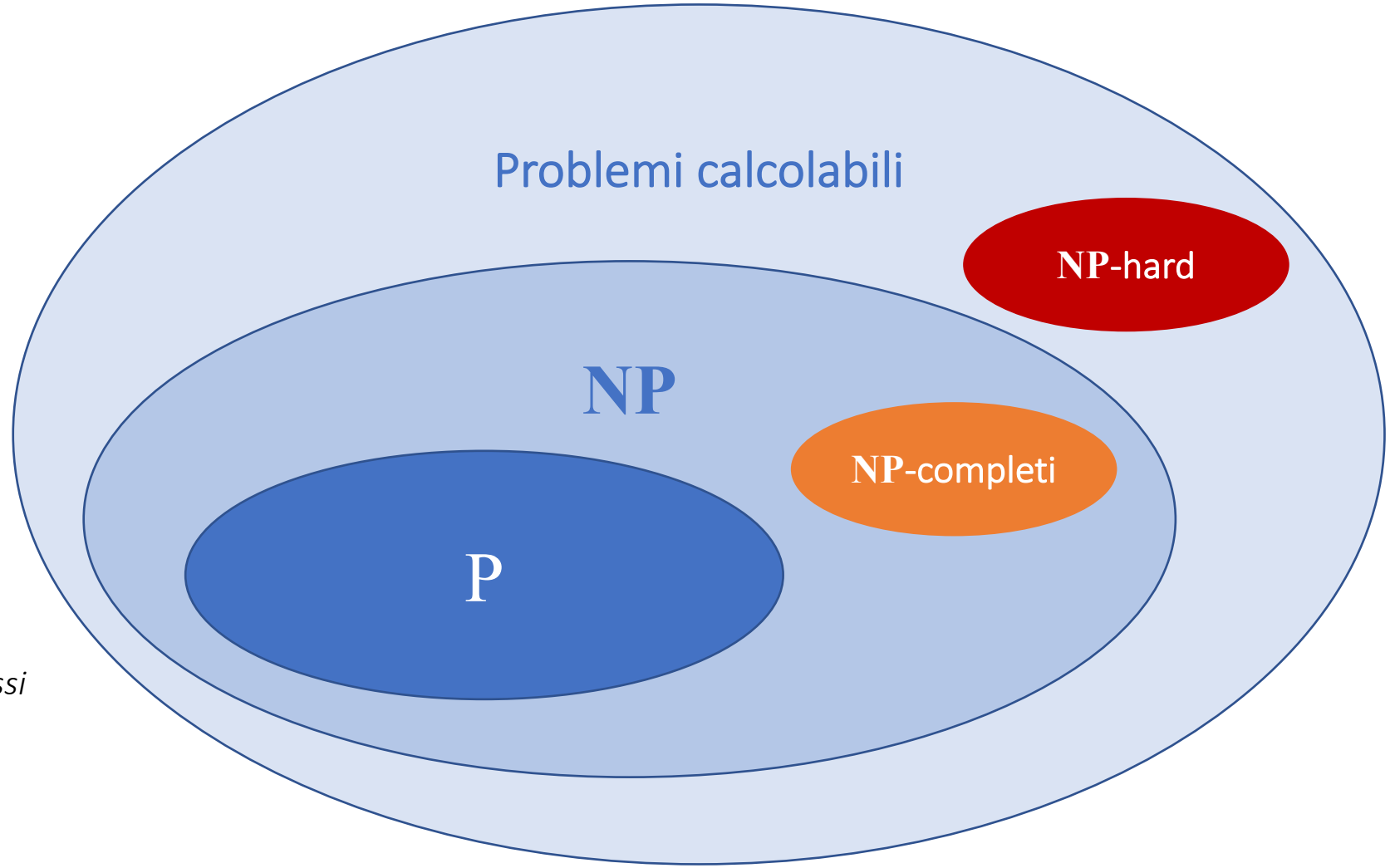
  La somma dell'insieme è $s + 2s - k + s + k = 4s$, per cui per risolvere PARTITION cerchiamo una partizione di somma $2s$. Esiste una soluzione di PARTITION (un sottoinsieme di $\{a_1, \ldots, a_n, b, c\}$ di somma $2s$) se e solo se esiste un sottoinsieme di $\{a_1, \ldots, a_n\}$ di somma k. Infatti, riordinando per semplicità gli elementi $a_1, \ldots, a_n$:

$$\underbrace{\overbrace{a_1 + \cdots + a_i}^{k} + \overbrace{a_{i+1} + \cdots + a_n}^{s-k}}_{s} + \overbrace{b}^{2s-k} + \overbrace{c}^{s+k} =$$

$$= \left(\overbrace{b}^{2s-k} + \overbrace{a_1 + \cdots + a_i}^{k} \right) + \left(\overbrace{a_{i+1} + \cdots + a_n}^{s-k} + \overbrace{c}^{s+k} \right)$$

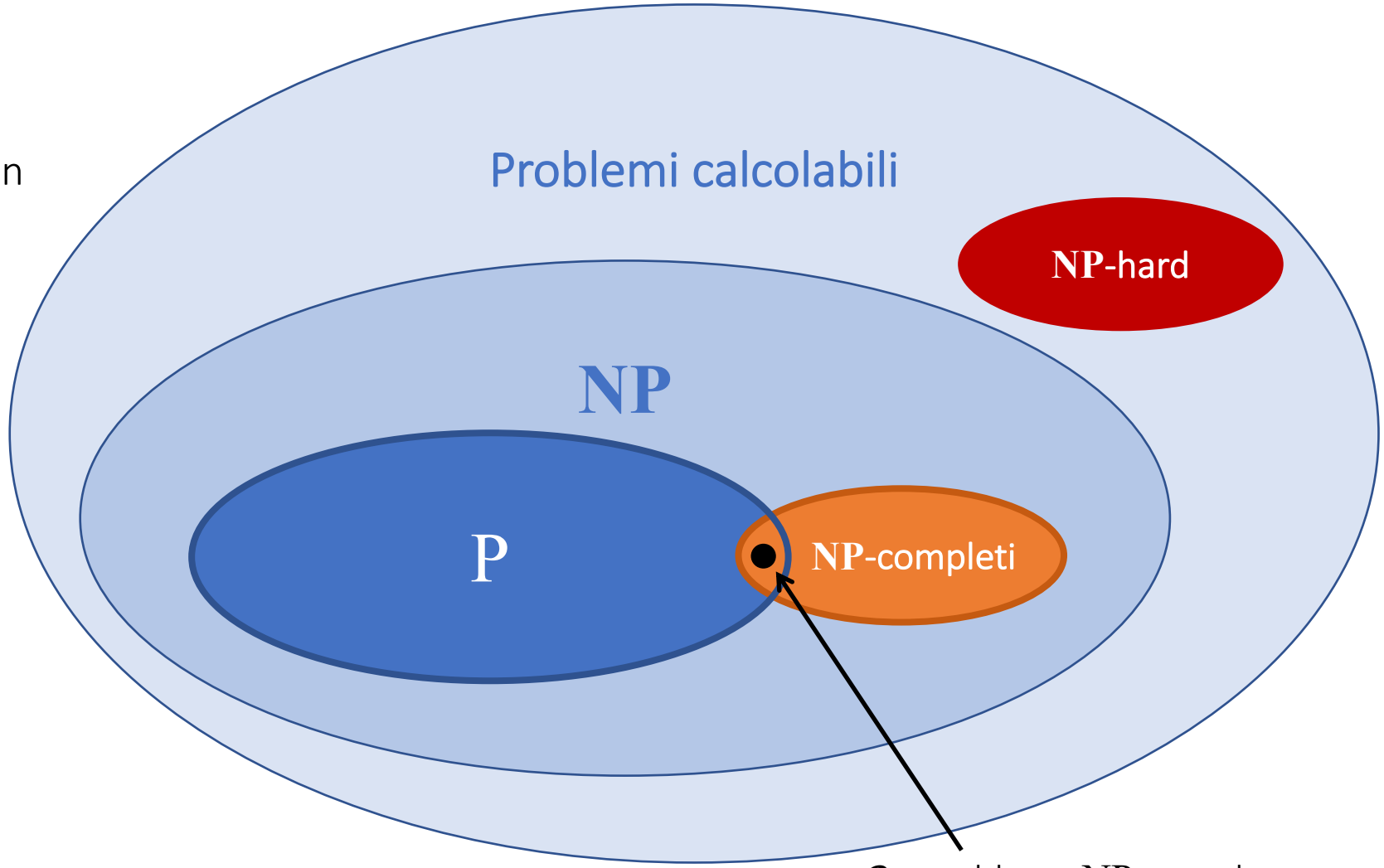
Classi di complessità

- Problemi **calcolabili**
  - *risolubili mediante un algoritmo*
- Problemi **NP**
  - *risolubili in tempo polinomiale mediante macchina non deterministica*
  - *verificabili in tempo polinomiale*
- Problemi **P**
  - *risolubili in tempo polinomiale*
- Problemi **NP-completi**
  - *sono NP e tutti i problemi NP sono riducibili in tempo polinomiale ad essi*
- Problemi **NP-hard**
  - *tutti i problemi NP sono riducibili in tempo polinomiale ad essi*



Il problema «P = NP»

- Gli insiemi P e NP sono distinti? I problemi in $NP \setminus P$ sono effettivamente non polinomiali?
- Se riuscissimo a dimostrare che un problema Q NP-completo è risolubile da un algoritmo di complessità polinomiale allora, siccome tutti i problemi NP sarebbero riducibili in tempo polinomiale a Q, tutti i problemi NP risulterebbero risolubili in tempo polinomiale e dunque $P \equiv NP$
- Il problema è tutt'ora aperto...



Applicazioni

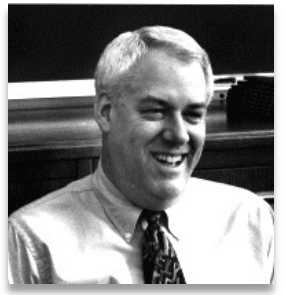
- Il fatto che il problema «P = NP» non sia ancora stato risolto, può essere sfruttato per interessanti applicazioni:
  - nell'ambito della crittografia: l'estrema difficoltà di decifrare un messaggio senza conoscere la chiave di cifratura a fronte della facilità con cui lo stesso messaggio può essere decifrato conoscendo la chiave

    l'algoritmo di decifratura, conoscendo la chiave, corrisponde all'algoritmo di verifica di una soluzione in tempo polinomiale; l'algoritmo di decifratura privo della conoscenza della chiave corrisponde all'algoritmo risolutivo, di elevata complessità computazionale (super-polinomiale)
  - nell'ambito dei protocolli di comunicazione sicuri di tipo «*zero knowledge*», in cui due interlocutori devono fidarsi reciprocamente (autenticarsi) senza scambiare un'informazione segreta nota ad uno solo dei due (come potrebbe essere una password)

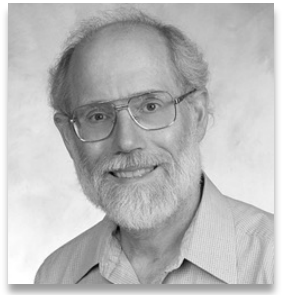
- Nell'ambito della crittografia (ormai indispensabile per la vita di tutti i giorni e per i modelli di interazione sociale, economica e politica a cui siamo abituati) la ricerca si sta spingendo verso algoritmi e metodi matematici di «*post-quantum cryptography*», dal momento che i cosiddetti «*computer quantistici*» possono, in linea teorica, essere equiparati alle «*macchine non deterministiche*», con cui possono essere risolti in tempo polinomiale, problemi attualmente risolubili in tempo super-polinomiale dai computer tradizionali

Riferimenti bibliografici

- Cormen, Leiserson, Rivest, Stein, «*Introduzione agli algoritmi e strutture dati*», terza edizione, McGraw-Hill (Cap. 2, Cap. 34)
- Michael R. Garey, David S. Johnson, «*Computer and Intractability. A Guide to the Theory of NP-Completeness*», Freeman and Company (1979)
- Fabrizio Luccio, «*La struttura degli algoritmi*», Bollati Boringhieri (1982)
- Marco Liverani, «*Qual è il problema? Metodi, strategie risolutive, algoritmi*», Mimesis (2005)
- Marco Liverani, *Dispense del Corso di Ottimizzazione Combinatoria: Algoritmi e complessità computazionale* (http://www.mat.uniroma3.it/users/liverani/doc/disp_oc_01.pdf)



Michael R. Garey
(1945)



David S. Johnson
(1945 – 2016)