Problemi facili, difficili e ... impossibili

Qualche parola sui concetti di calcolabilità e complessità computazionale

Marco Liverani

marco.liverani@uniroma3.it

LESSENZIALE2 – Pesaro, 11-13 ottobre 2024

L’essenziale (a mio giudizio)

- Da informatico e da insegnante di informatica, ormai da tempo non penso più che l’*essenziale* sia un **computer potente** o un bel **programma**
- A lungo ho pensato che l’*essenziale* (l’essenza del mio mestiere) fosse la progettazione di un **algoritmo**, efficiente ed elegante
- Oggi penso invece che l’*essenziale*, ciò che mi piace portare con me e raccontare a chi ha voglia di ascoltarmi, siano i concetti di **calcolabilità** e di **complessità computazionale**

... certo, sono parole complicate, non amichevoli,

indubbiamente legate ai computer e agli algoritmi, eppure ...

Semplificare può essere «scivoloso»...

«Ogni tanto è necessario dire delle cose difficili,
ma bisognerebbe dirle nel modo più semplice di cui si è capaci.»

– Godfrey H. Hardy

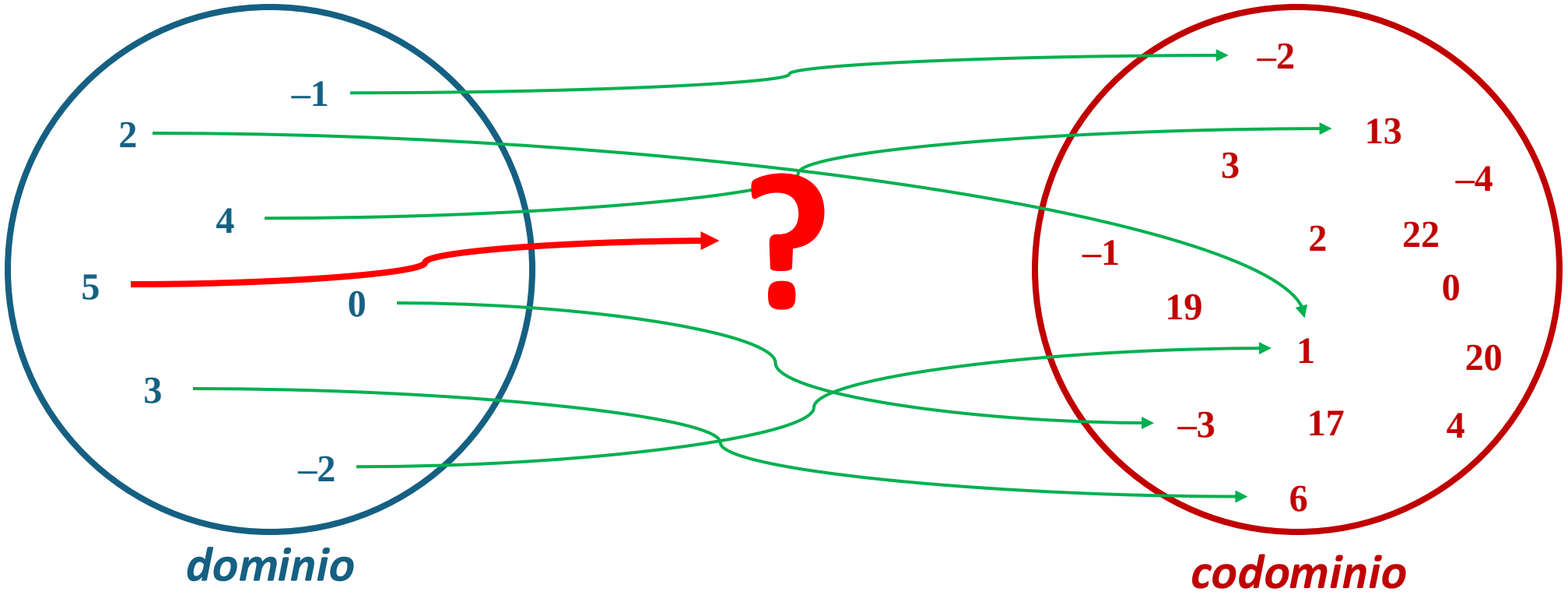
Godfrey H. Hardy
(1877–1947)

Algoritmi

- L'**algoritmo** è un concetto che spesso viene associato direttamente all'informatica, all'ambito della programmazione dei computer
- È un'interpretazione ***corretta***, ma è anche un'interpretazione *limitata*
- Il concetto di algoritmo va ricondotto alla capacità di **descrivere un procedimento di calcolo** che ci permetta, attraverso la concatenazione di operazioni elementari, di risolvere un problema
- E quindi è un concetto che possiamo associare direttamente alla «***calcolabilità***» di un problema, ossia la possibilità di calcolare in modo effettivo, in un tempo finito (possibilmente breve), le soluzioni di ogni possibile istanza di un problema
- È un concetto che possiamo anche associare a quello di **funzione in matematica**

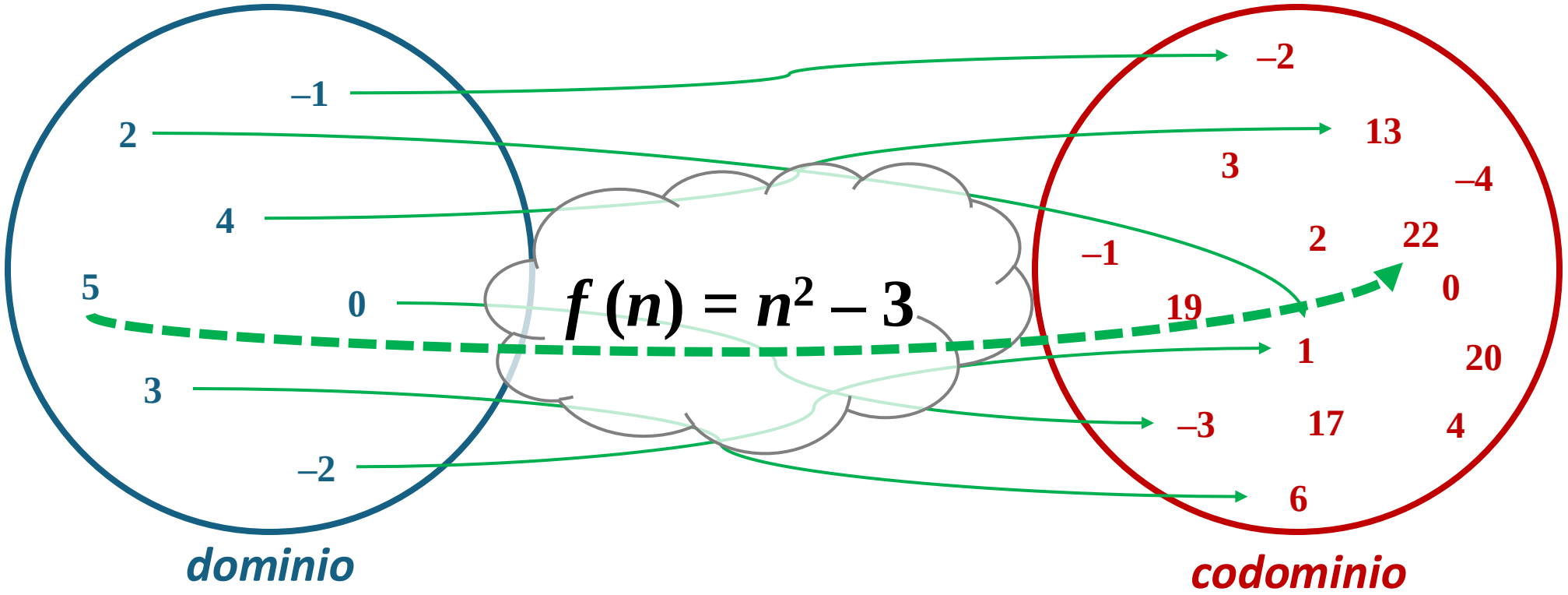
Funzioni

Una funzione è una **relazione tra due insiemi** che associa ad *ogni* elemento del **dominio** *uno ed un solo* elemento del **codominio**



Funzioni

Possiamo **descrivere la funzione** (la relazione tra insiemi) in modo sintetico attraverso un'espressione che consenta a «*chiunque*», per qualsiasi valore del dominio, di calcolare il corrispondente valore del codominio



Funzioni ed espressioni

- La funzione viene descritta «***raccontando***» il procedimento con cui è possibile associare un elemento del dominio della funzione a un elemento del codominio
- L'**espressione aritmetica** è una *formulazione convenzionale*, molto compatta/sintetica, che si basa su un insieme di simboli, su una ***sintassi*** ben precisa e naturalmente su una ***semantica*** che ci permette di dare un significato all'espressione
- L'espressione aritmetica adotta quindi un ***linguaggio*** utilizzato per esprimere un procedimento di calcolo matematico

$$f(n) = n^2 - 3$$

prendi un numero n dal dominio della funzione

moltiplica n per se stesso

al risultato che hai ottenuto, sottrai 3

hai ottenuto così l'elemento del codominio che corrisponde a n

Prima parentesi

Sintassi e semantica delle espressioni aritmetiche

Espressioni e numeri

- Il **linguaggio** delle espressioni aritmetiche e la **semantica** che esse nascondono si basano anche sul modo in cui rappresentiamo i numeri
- Il sistema posizionale dei «numeri arabi» è ben diverso dal sistema numerico romano o da quello greco: i romani, infatti, non usavano quel sistema di numerazione per eseguire i calcoli, ma utilizzavano un abaco
- Oggi in tutto il mondo i calcoli numerici sono basati sul sistema posizionale e sui numeri arabi, con una notazione in «base 10» ma trovano ancora spazio altri sistemi numerici come ad esempio il sistema «sessagesimale» (in base 60!) che risale alla civiltà babilonese
  - Lo ritroviamo nell'espressione dei gradi di un angolo o nella misurazione del tempo (60 secondi formano un minuto, 60 minuti un'ora, ...)
  - Ho sentito dire al prof. Odifreddi (ma lui dove l'ha letto?) che derivano dall'uso delle 12 falangi di quattro dita di una mano: il pollice si usa per contare fino a 12 toccando le falangi delle altre dita; le dita dell'altra mano contano il numero di «dodicine»: $12 \times 5 = 60$
- Inutile ricordare che invece i calcolatori digitali operano con numeri in «base 2»

Sintassi e semantica delle espressioni

- I sistemi numerici determinano la nostra capacità di eseguire in modo semplice o estremamente complicato, delle operazioni, dei calcoli
- Fin dalla scuola primaria siamo abituati a scrivere le espressioni in un certo modo
- Si tratta di una notazione «**infissa**», che utilizza degli operatori binari, disponendo gli operandi a sinistra e a destra dell'operatore e dando un significato diverso alla diversa posizione dell'operando rispetto all'operatore (es.: il numeratore e il denominatore di una divisione)
- Le **parentesi** sono usate per dare una priorità di calcolo alle diverse componenti dell'espressione (es.: $3 - 2 \times 4 \neq (3 - 2) \times 4$)

- *Non penserete mica che questo sia l'unico modo di formulare un'espressione aritmetica? O che sia il più «semplice»? Temo sia solo una questione di abitudine...*

Sintassi e semantica delle espressioni

- La notazione polacca inversa è una ***notazione postfissa*** che semplifica la scrittura di espressioni aritmetiche, evitando l'uso delle parentesi per dare priorità agli operatori
- Alla base del procedimento di rappresentazione della notazione c'è la seguente sintassi:

operando operando operatore

Ossia, invece di scrivere «3 + 2» si scriverà «3 2 +»

- L'idea è quella di leggere l'espressione da sinistra a destra (come al solito) e di sostituire l'operatore e i due operandi che lo precedono con il valore dell'espressione calcolata, ripetendo il procedimento iterativamente fino a quando non rimarrà soltanto il risultato:

«3 × (2 + (12 – 4) / 2)» → «12 4 – 2 / 2 + 3 ×»

12 4 – 2 / 2 + 3 × → **8 2 /** 2 + 3 × → **4 2 +** 3 × → **6 3 ×** → **18**

- I linguaggi dei computer convertono le espressioni «infisse» in espressioni «postfisse» in RPN per calcolarne il risultato: il procedimento di calcolo risulta più semplice

Sintassi e semantica delle espressioni

- L'algebra moderna ci fornisce un linguaggio potente ed estremamente compatto per rappresentare espressioni matematiche anche molto complesse
- La rappresentazione dei numeri e la codifica delle espressioni sono due strumenti, non soltanto formali/convenzionali, in grado di guidare anche la nostra capacità di eseguire in modo efficiente i calcoli necessari per giungere alla soluzione del problema
- Non è sempre stato così: al di là dei complicatissimi sistemi numerici utilizzati nel passato, che già di per sé hanno agevolato (come nel caso degli arabi, dei babilonesi e degli indiani) o ostacolato (come nel caso dei numeri romani) il calcolo numerico e l'astrazione matematica, la strada per giungere alla notazione odierna è stata assai lunga
- Il matematico italiano **Raphael Bombelli** (1526 – 1572) scriveva, ad esempio, nel suo trattato «Algebra – Libro III»:

  Trovinsi due numeri che siano in proportione come 3 e 4 e che moltiplicato il minore per 5 e il maggiore per 2, li produtti gionti insieme faccino 46

  Oggi scriveremmo semplicemente: trovare x tale che $3x \times 5 + 4x \times 2 = 46 \quad \Rightarrow \quad x = 2$

Chiusa parentesi

Funzioni e algoritmi

- L'espressione aritmetica con cui descriviamo la corrispondenza tra elementi del dominio ed elementi del codominio descrive un «procedimento» per calcolare l'elemento del codominio corrispondente ad ogni elemento del dominio
- Un algoritmo è esattamente questo:
  - la descrizione di un procedimento di calcolo
  - con una successione di passi/operazioni elementari
  - che consenta calcolare la soluzione del problema eseguendo un numero finito di operazioni
- Vediamo un esempio elementare:

$$f : \mathbb{R} \to \mathbb{R}$$

$$f(x) = x^2 - 3x + 5$$

Algoritmo $f(x)$:

1. $a = x \times x$
2. $b = 3 \times x$
3. $c = a - b + 5$
4. restituisci c

una sequenza di operazioni

Funzioni e algoritmi

- L'espressione aritmetica con cui descriviamo la corrispondenza tra elementi del dominio ed elementi del codominio descrive un «procedimento» per calcolare l'elemento del codominio corrispondente ad ogni elemento del dominio
- Un algoritmo è esattamente questo:
  - la descrizione di un procedimento di calcolo
  - con una successione di passi/operazioni elementari
  - che consenta calcolare la soluzione del problema eseguendo un numero finito di operazioni
- Vediamo un esempio elementare:

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

$$f(n) = 1 + 2 + 3 + \cdots + n = \sum_{k=1}^{n} k$$

Algoritmo $f(n)$:

1. $s = 0$
2. $k = 1$
3. $s = s + k$ — *una iterazione di*
4. $k = k + 1$ — *operazioni*
5. se $k \leq n$ vai al passo 3 altrimenti prosegui
6. restituisci s

Funzioni e algoritmi

- L'espressione aritmetica con cui descriviamo la corrispondenza tra elementi del dominio ed elementi del codominio descrive un «procedimento» per calcolare l'elemento del codominio corrispondente ad ogni elemento del dominio
- Un algoritmo è esattamente questo:
  - la descrizione di un procedimento di calcolo
  - con una successione di passi/operazioni elementari
  - che consenta calcolare la soluzione del problema eseguendo un numero finito di operazioni
- Vediamo un esempio elementare:

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

$$f(n) = \begin{cases} n/2 \text{ se } n \text{ è pari} \\ 3n+1 \text{ se } n \text{ è dispari} \end{cases}$$

Algoritmo $f(n)$:

1. se n è pari
2. allora $r = n / 2$
3. altrimenti $r = 3n + 1$
4. restituisci r

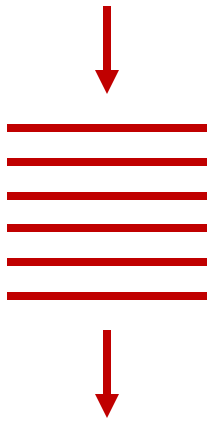
una struttura condizionale (o alternativa)

Seconda parentesi

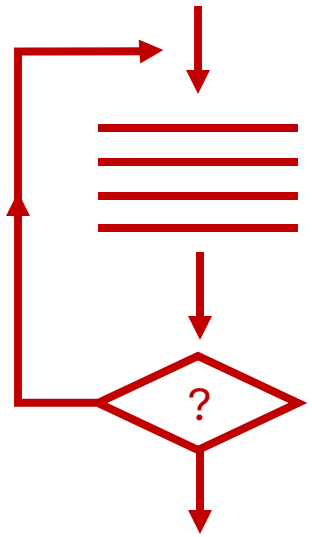
L’uso delle parentesi nelle espressioni aritmetiche e le regole della «programmazione strutturata»

Programmazione strutturata

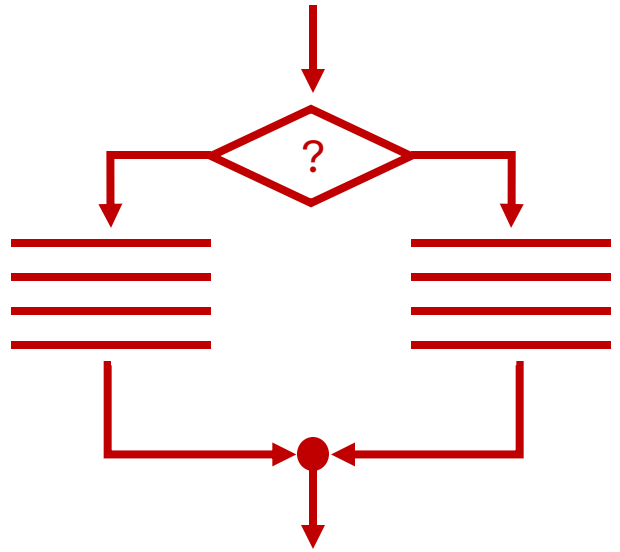
I tre esempi che abbiamo appena visto rappresentano tre strutture algoritmiche fondamentali:



sequenziale



iterativa



condizionale

Programmazione strutturata

- La «***programmazione strutturata***» richiede che qualsiasi algoritmo sia progettato utilizzando **solo le tre strutture algoritmiche sequenziale**, **iterativa** e **condizionale**, che possono essere ***concatenate*** fra di loro o ***nidificate*** una dentro l'altra, ma non possono mai «***accavallarsi***»
- È evidente l'analogia con la corretta **apertura e chiusura delle parentesi** in una espressione aritmetica
  - è il motivo per cui ai ragazzi delle scuole medie si insegna ad usare parentesi di forma differente (*tonde*, *quadre* e *graffe*) per evidenziare che le espressioni tra parentesi possono essere ***concatenate*** o ***nidificate*** fra loro (completamente contenute una nell'altra), ma non possono ***accavallarsi***:

$$a \times \{ [\, b + c \times (a - d) \,] : [\, 7 \times (d + e) \,] \}$$

- Il Teorema Fondamentale della Programmazione Strutturata (Böhm & Jacopini, 1966) dice che attenersi alle regole della programmazione strutturata non è certo un limite: afferma che qualsiasi problema sia risolubile per via algoritmica, può anche essere risolto con un algoritmo che rispetti le regole della programmazione strutturata


Chiusa parentesi

Funzioni e algoritmi

- Un **algoritmo** non è altro che una descrizione di una procedura di calcolo per **passi elementari**, che produca **in un tempo finito** una soluzione al problema, per ogni istanza del problema
- Vediamo un altro esempio (questa volta *sbagliato!*):

Consideriamo $f(n) = \sqrt{n}$

$\forall\, n \in \mathbb{N}$ risulta $k = \sqrt{n} \iff k^2 = n$

Algoritmo $f(n)$:
1. $k = 0$
2. $k = k + 1$
3. se $k^2 \neq n$ vai al passo 2 altrimenti prosegui
4. restituisci k

Se $n = 9$:

$$k = 1 \Rightarrow k^2 = 1, k = 2 \Rightarrow k^2 = 4, \boxed{k = 3 \Rightarrow k^2 = 9}$$

Se invece $n = 8$ ...

$$k = 1 \Rightarrow k^2 = 1, k = 2 \Rightarrow k^2 = 4, k = 3 \Rightarrow k^2 = 9, \ldots$$

non finisce mai e non trova la soluzione per $n = 8$

Funzioni calcolabili

- Siamo arrivati al dunque:

Una funzione è quindi **calcolabile** se esiste un **algoritmo** che descriva come mettere in corrispondenza ogni valore del dominio della funzione con un elemento del codominio

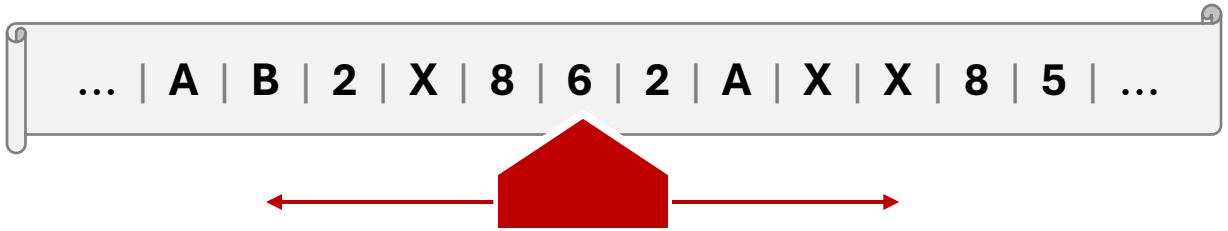
- La calcolabilità si preserva attraverso:
  - la ***composizione*** di funzioni (la concatenazione di algoritmi)
    se $f : A \to B$ e $g : B \to C$ sono funzioni calcolabili, allora anche $h = g \circ f$ $(h(n) = g(f(n))$ è calcolabile
  - la ***ricorsione*** (le strutture algoritmiche iterative): $f(0) = k, \quad f(n) = g(n, f(n-1))$
  - es.: $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$

    quindi: $\begin{cases} n! = 1 \text{ se } n = 1 \\ n! = n \times (n-1)! \text{ se } n > 1 \end{cases}$

Modelli di calcolo e calcolabilità

Alan Turing
(1912 – 1954)

- Per definire in modo essenziale il concetto di algoritmo nella prima metà del ‘900 furono proposti diversi modelli di calcolo, tutti fra loro equivalenti
- Tra questi è celeberrima la **Macchina di Turing** (MdT) proposta da **Alan Turing** nel 1936 (ben prima che vedessero la luce i primi rudimentali computer!)
- È un modello «astratto» di macchina di calcolo, non realizzabile nella pratica (non c’entra nulla con gli ingegnosi macchinari che Turing realizzò durante la Seconda Guerra Mondiale per decifrare i codici segreti dei nazisti!)
- È basato su un **nastro infinito** su cui possono essere scritti e letti dalla macchina un numero arbitrario di **simboli**; l’algoritmo si basa sullo **scorrimento del nastro** verso destra e verso sinistra del, sulla **lettura** e **scrittura** di simboli e sul **cambiamento di stato**



Modelli di calcolo e calcolabilità

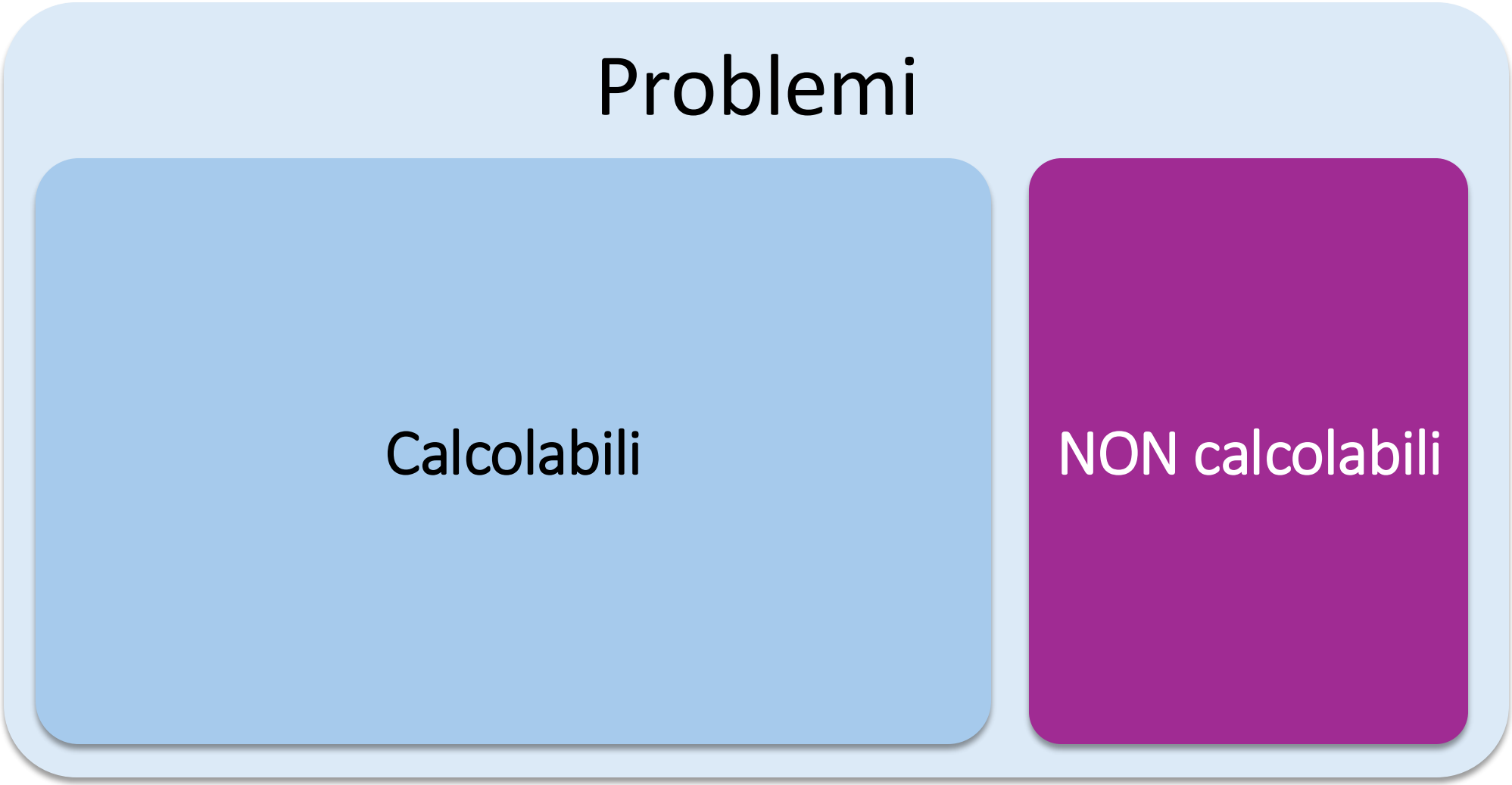
Il risultato più importante ottenuto da Turing grazie al suo modello di calcolo possiamo sintetizzarlo nei seguenti punti:

- un problema è calcolabile se e solo se è calcolabile con una MdT
- le MdT possono essere enumerate e associate ai numeri naturali
- è possibile definire una MdT Universale, che acquisisca in input una MdT e la «esegua»
- non è possibile definire una MdT Universale che, acquisita in input una MdT definita per un determinato problema, sia in grado di stabilire a priori se la MdT risolve il problema per ciascuna istanza

Quest'ultimo è il cosiddetto «**problema della fermata**»

Ci dice che **esistono problemi la cui soluzione non può essere calcolata automaticamente** (con una MdT e quindi con un qualsiasi algoritmo)

Problemi calcolabili e non calcolabili



Non tutto ciò che è calcolabile può essere calcolato

- Abbiamo stabilito il concetto di **calcolabilità di un problema**: la capacità di esprimere un **procedimento di calcolo deterministico** per associare ad ogni elemento del dominio di una funzione (input) un elemento del codominio (output)
- Abbiamo stabilito che esistono funzioni (problemi) **non calcolabili**

- Per chi si occupa di algoritmi questo pone dei limiti, definisce un perimetro per focalizzarsi su quei problemi che sono calcolabili

- Limitandoci ai soli problemi calcolabili, possiamo affermare che tutti i problemi calcolabili siano *effettivamente calcolabili* con gli strumenti di calcolo che abbiamo a disposizione oggi?
- Si apre così un altro scenario relativo allo studio della **complessità computazionale dei problemi calcolabili**

Svuotare l'oceano con un cucchiaio

- È possibile svuotare un bicchiere d'acqua con un cucchiaino da caffè?
  - *Certamente! Basta un po' di pazienza, ma in pochi minuti il gioco è fatto*

- È possibile svuotare una pentola piena d'acqua con un cucchiaino da caffè?
  - *Beh, sì, certo. Occorre maggiore pazienza, ma in qualche ora si può fare…*

- È possibile svuotare l'oceano con un cucchiaino da caffè?
  - *In linea teorica (a meno dei processi metereologici che potrebbero aumentare o diminuire la quantità di acqua presente nell'oceano) sembrerebbe di poter rispondere di sì, ma il tempo richiesto per portare a termine l'operazione potrebbe richiedere centinaia di anni o forse più…*

- Il problema è <u>calcolabile</u>, ma la procedura per risolverlo è <u>impraticabile</u>

Complessità di un algoritmo

- La complessità computazionale di un algoritmo è una misura della sua efficienza nell'uso del tempo o delle risorse di calcolo che ha a disposizione (es.: la memoria del computer)
- Limitiamoci alle performance nel tempo: il calcolo della complessità di un algoritmo si basa sull'assunto che **per eseguire una singola operazione elementare l'esecutore dell'algoritmo impieghi sempre lo stesso tempo** (*non sono previste la noia e la stanchezza*)
- Allora calcolare «quanto tempo» impiega un algoritmo per risolvere una determinata istanza di un problema, equivale a **contare il numero di operazioni elementari che deve eseguire** per calcolare la soluzione
  - naturalmente si impiega più tempo (si devono compiere più operazioni) per risolvere un problema in cui si debbano elaborare tanti dati, piuttosto che uno in cui ci siano pochi dati da elaborare
  - es.: è chiaro che impiego più tempo (devo svolgere più operazioni) per trovare l'elemento massimo in un insieme di 10.000 numeri che in un insieme di 10 numeri; ma di fatto sono due ***istanze*** di ***dimensione diversa*** dello stesso identico problema (che si risolvono con lo stesso algoritmo)

Problemi facili e difficili

- Occupiamoci d'ora in avanti solo di **problemi calcolabili**: d'altra parte non è molto proficuo occuparsi di quelli *non* calcolabili (impossibili da risolvere per via algoritmica)

- È possibile stabilire se un problema è **più facile** o **più difficile** di un altro?
  - Esempio: *è più facile calcolare un integrale o una derivata? Ah, beh, dipende…*

- Gli informatici teorici si sono dati una risposta che porta a delle speculazioni assai interessanti e ricche di conseguenze

C’è problema e problema

Tentiamo innanzi tutto una classificazione dei problemi sulla base del tipo di risposta, del tipo di soluzione che chiedono di costruire:

- Problemi di **decisione** (o di esistenza)
  - richiedono di decidere se una soluzione **esiste**
  - la risposta è «binaria»: *sì* o *no*, *vero* o *falso*, *esiste* o *non esiste*
  - es.: è possibile dividere per due il numero n? Esiste uno zero della funzione $f(x)$ nell’intervallo $[a, b]$? Esiste una strada per andare con i mezzi pubblici da casa a scuola? Ecc.

- Problemi di **ricerca** (di una soluzione)
  - richiedono di **esibire una soluzione**, se esiste
  - la soluzione può essere trovata se il problema di decisione corrispondente ammette una soluzione
  - es.: trovare un numero che possa dividere n; trovare un valore di $x \in [a, b]$ tale che $f(x) = 0$; trovare un percorso per andare da casa a scuola con i mezzi pubblici, ecc.

C’è problema e problema

Tentiamo innanzi tutto una classificazione dei problemi sulla base del tipo di risposta, del tipo di soluzione che chiedono di costruire:

- Problemi di **enumerazione** (tutte le soluzioni)
  - richiedono di trovare non una soltanto, ma **tutte** le soluzioni ammesse dal problema
  - di fatto questo genere di problema è un’estensione del corrispondente problema di ricerca
  - es.: trovare tutti i numeri che dividono n; trovare tutti i valori di $x \in [a, b]$ tali che $f(x) = 0$; trovare tutti i possibili percorsi per andare da casa a scuola con i mezzi pubblici, ecc.

- Problemi di **ottimizzazione** (le soluzioni «migliori»)
  - richiedono di trovare, tra le soluzioni ammissibili, quelle che **minimizzano o massimizzano un determinato criterio**, espresso mediante la «*funzione obiettivo*» per il problema
  - sono una sorta di raffinamento del problema di enumerazione, perché trovare tutte le soluzioni ammissibili e tra queste scegliere le migliori, spesso non è conveniente (è troppo laborioso)
  - es.: trovare il divisore di n **più grande**; trovare i valori di $x \in [a, b]$ tali che $f(x) = 0$ e che al tempo stesso **rendano minima $\varphi(x)$**; trovare il percorso per andare da casa a scuola con il minor numero di cambi di mezzi pubblici, ecc.

Problemi facili o difficili (da calcolare)

- Indichiamo con $f_A(n)$ il numero di operazioni elementari che devono essere eseguite per risolvere con l'algoritmo A una istanza del problema caratterizzata da n informazioni da elaborare; $f_A(n)$ è la **complessità computazionale** dell'algoritmo A
- Se riusciamo ad escogitare un algoritmo B per risolvere lo stesso problema, tale che $f_B(n) < f_A(n)$, allora possiamo dire che l'algoritmo B è più efficiente (è più veloce) dell'algoritmo A
- Se riusciamo a dimostrare che **il problema non può essere risolto in modo più efficiente di quanto faccia l'algoritmo B**, allora la funzione $f_B(n)$ indica la complessità computazionale del problema
- In questo modo possiamo confrontare fra loro problemi completamente diversi e, sotto questo particolare punto di vista, quello della calcolabilità, possiamo **stabilire se un problema è effettivamente più facile e più difficile di un altro**

La complessità è una funzione

- Chiedersi quanto tempo impiega un algoritmo per calcolare la soluzione di un problema è improprio: il tempo impiegato (il numero di operazioni svolte) dipende quasi sempre dalla specifica istanza del problema che si intende risolvere
- O meglio: dipende dalla dimensione dell'istanza del problema, dal numero di dati / parametri / informazioni che dobbiamo elaborare per calcolare la soluzione
- Esempio: l'algoritmo di ordinamento «*selection sort*» per ordinare una sequenza di n numeri

Algoritmo **SelectionSort**(a_1, ..., a_n):
1. per $i = 1, 2, ..., n - 1$ ripeti:
2. per $j = i + 1, i + 2, ..., n$ ripeti:
3. se $a_i > a_j$ allora scambia a_i e a_j
4. fine ciclo
5. fine ciclo

2	4	5	7	9
a_1	a_2	a_3	a_4	a_5

questa operazione viene eseguita $(n-1) + (n-2) + ... + 1$ volte

quindi $\frac{n \times (n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$

La complessità è una funzione

- Stimare esattamente la complessità computazionale di un algoritmo è difficile: lo stesso algoritmo può essere riscritto in modi differenti, che possono modificare (di poco) la sua complessità computazionale
- Lo stesso algoritmo può essere valutato diversamente, ad esempio $\frac{3}{2}n^2 - \frac{3}{2}n$
- Ciò che non cambia è l'ordine di grandezza della funzione con cui esprimiamo la complessità dell'algoritmo: il suo andamento asintotico, in entrambi i casi sono dei **polinomi di grado 2**

Algoritmo **SelectionSort**(a_1, ..., a_n):
1. per $i = 1, 2, ..., n - 1$ ripeti:
2. per $j = i + 1, i + 2, ..., n$ ripeti:
3. se $a_i > a_j$ allora scambia a_i e a_j
4. fine ciclo
5. fine ciclo

questa operazione viene eseguita $(n - 1) + (n - 2) + ... + 1$ volte

quindi $\frac{n \times (n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$

circa n^2 operazioni

Qualche esempio

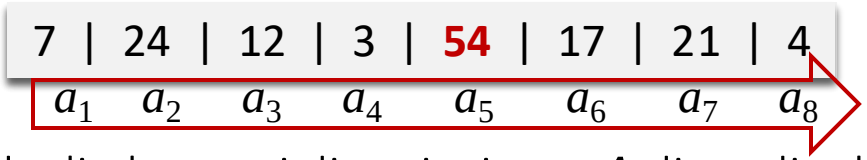
- I problemi matematici / numerici hanno degli algoritmi facili per calcolare la soluzione, spesso hanno una complessità computazionale costante o lineare

- Esempio: trovare x tale che $a\,x^2 + b\,x + c = 0$ è un problema che si risolve in un numero costante di operazioni, indipendentemente dai valori di a, b e c

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Qualche esempio

- Per trovare qualche problema semplice, ma con algoritmi più complessi possiamo considerare problemi di combinatoria sugli insiemi finiti
  - trovare l'elemento massimo su un insieme A di cardinalità n richiede di compiere n operazioni

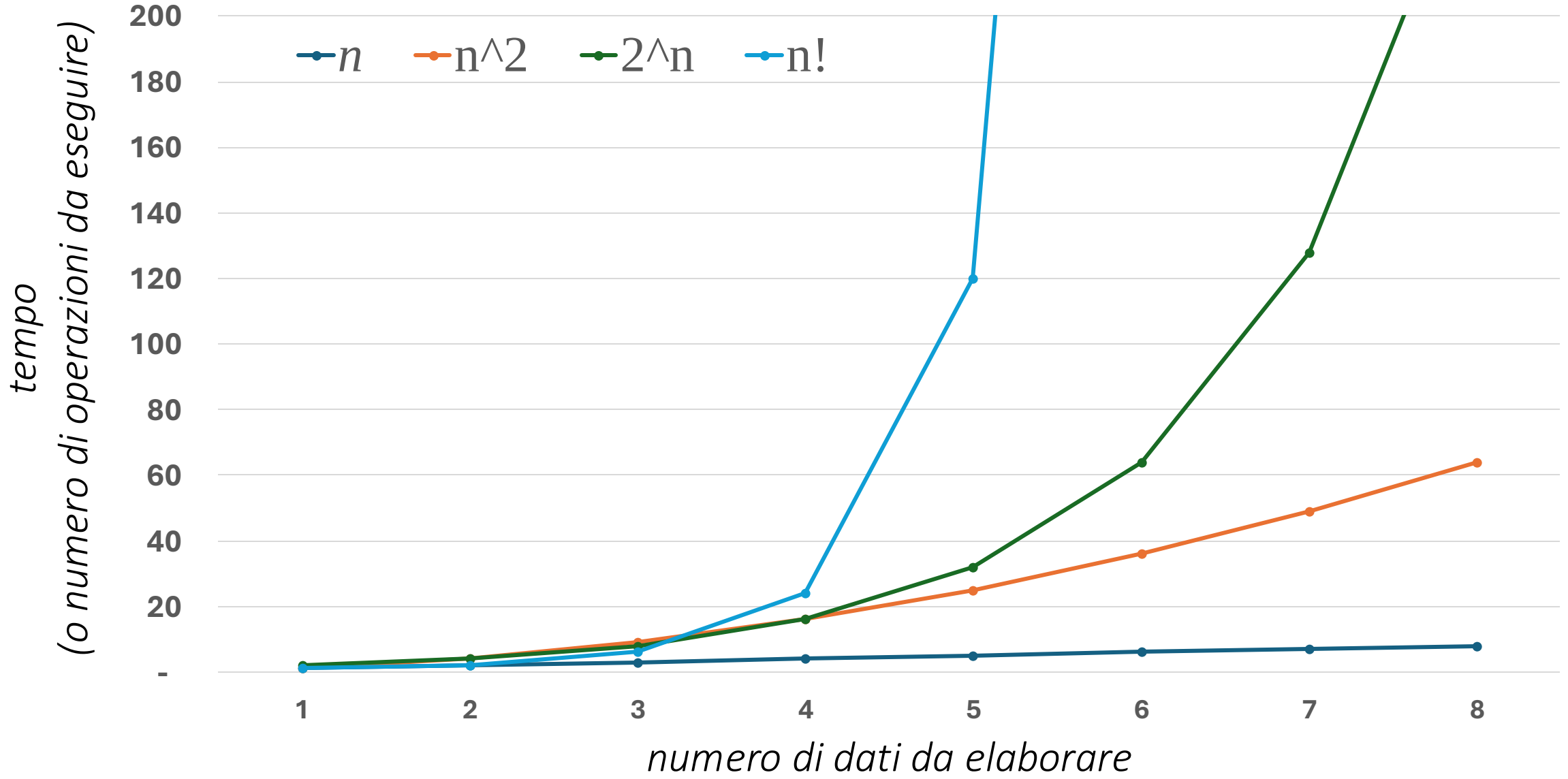


  - trovare la permutazione degli elementi di un insieme A di cardinalità n, tale che gli elementi risultino ordinati in ordine crescente, richiede di compiere n^2 operazioni (ma si può risolvere in $n \log_2 n$ operazioni!)

  - calcolare tutti i sottoinsiemi dell'insieme A di cardinalità n richiede di compiere almeno 2^n operazioni

  - calcolare tutte le permutazioni degli elementi dell'insieme A di cardinalità n richiede di compiere almeno $n!$ operazioni (n fattoriale: $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$)

Qualche esempio



Crescita asintotica

- Indichiamo con $O(f(n))$ la classe di complessità degli algoritmi
- Più è «bassa» la complessità dell'algoritmo, più questo è efficiente
- Cosa significa in termini pratici? Supponiamo di utilizzare un computer che esegua **1.000.000 di operazioni al secondo** (*spoiler: è un computer molto lento*...)

n	$O(n^2)$	$O(n^3)$	$O(n^5)$	$O(2^n)$
10	0 sec.	0,001 sec.	0,1 sec.	0 sec.
20	0 sec.	0,008 sec.	3,2 sec.	1 sec.
30	0,001 sec.	0,027 sec.	24 sec.	18 min.
40	0,002 sec.	0,064 sec.	102 sec.	305 ore
50	0,003 sec.	0,125 sec.	313 sec.	13.031 giorni
60	0,004 sec.	0,216 sec.	13 min.	36.559 anni
70	0,005 sec.	0,343 sec.	28 min.	374.363 secoli
100	0,010 sec.	1 sec.	2,78 ore	40 milioni di miliardi di anni

L'ultimo computer Apple con CPU «M3 MAX» immesso sul mercato esegue **18.000 miliardi** di operazioni al secondo

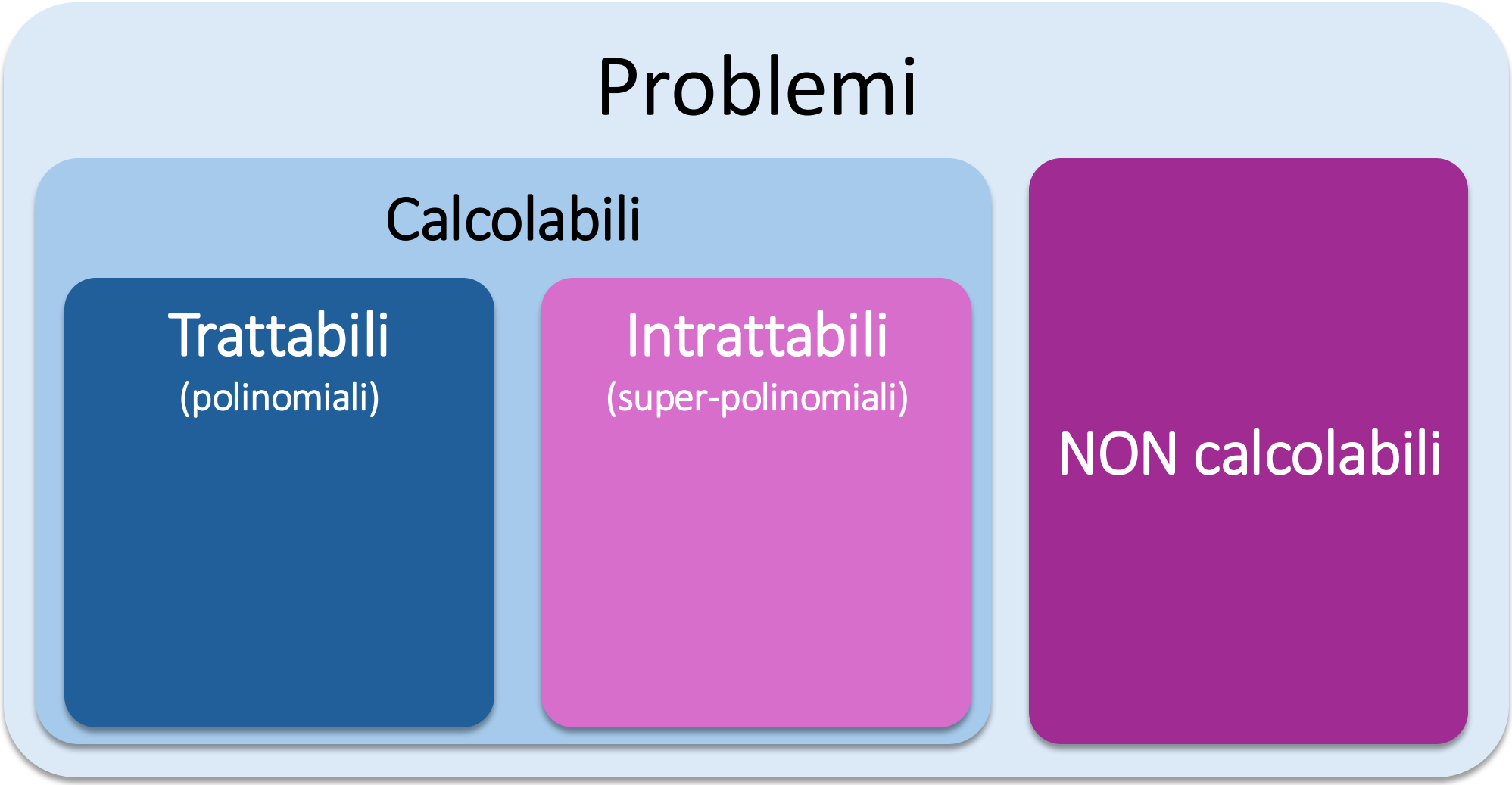
Per risolvere l'istanza $n = 100$ con l'algoritmo di complessità $O(2^n)$ impiega **solo 2 secoli** ...

Ma per l'istanza $n = 125$ impiega comunque **74 milioni di miliardi di anni** ...

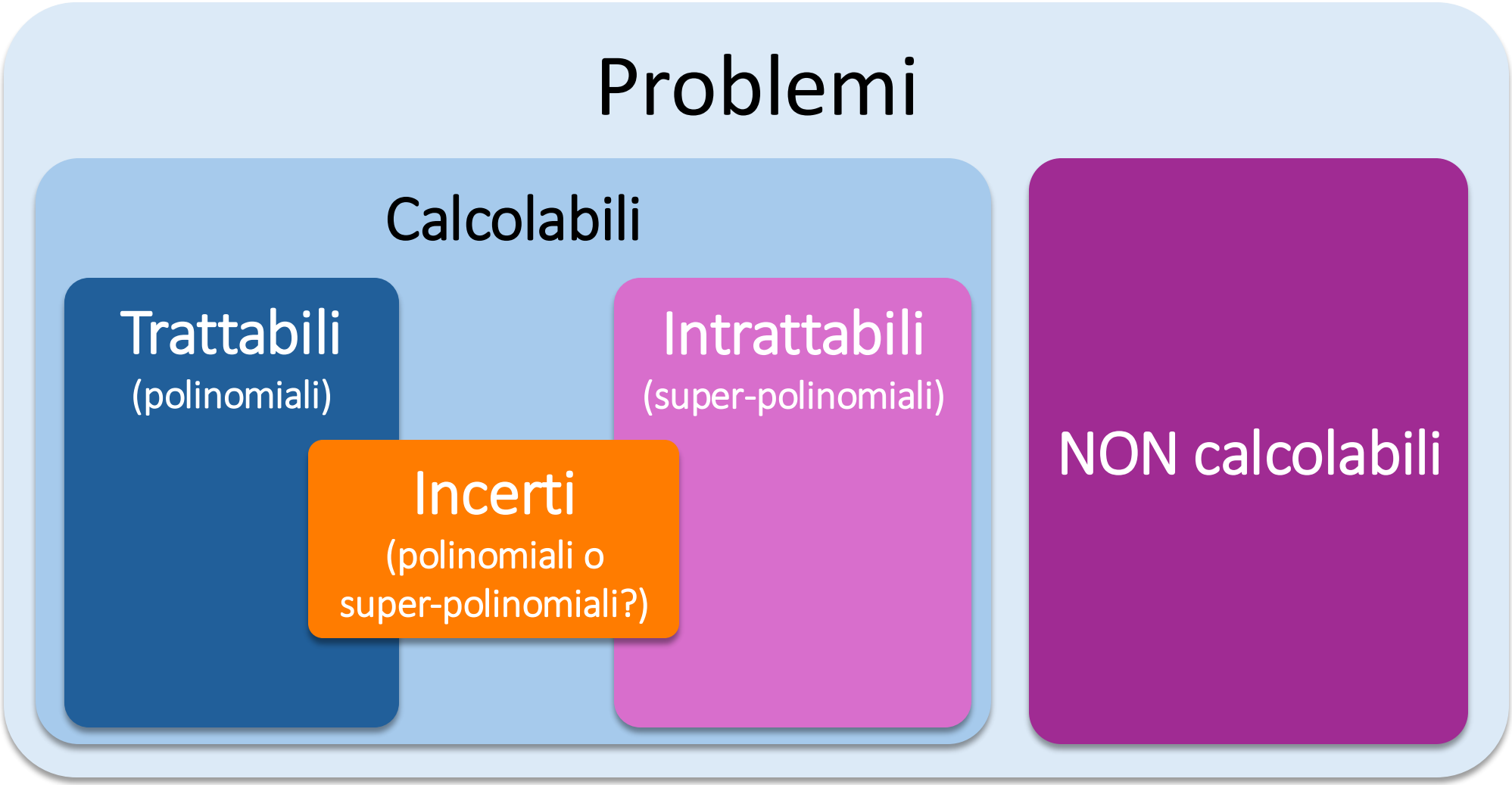
Problemi calcolabili, ma intrattabili

- È chiaro che pur aumentando la potenza di calcolo disponibile, determinati problemi risultano comunque ***intrattabili***
- Il tempo richiesto per calcolare la soluzione, pur essendo finito, è talmente grande che non ha senso approcciare la soluzione di simili problemi con un procedimento algoritmico
- Risultano trattabili i problemi che ammettono come miglior algoritmo risolutivo un algoritmo di complessità polinomiale: chiamiamo questo insieme di problemi classe dei problemi polinomiali ($\mathbf{P}$)

Problemi calcolabili, non calcolabili e intrattabili



Non finisce qui…



Intrattabile = difficilissimo?

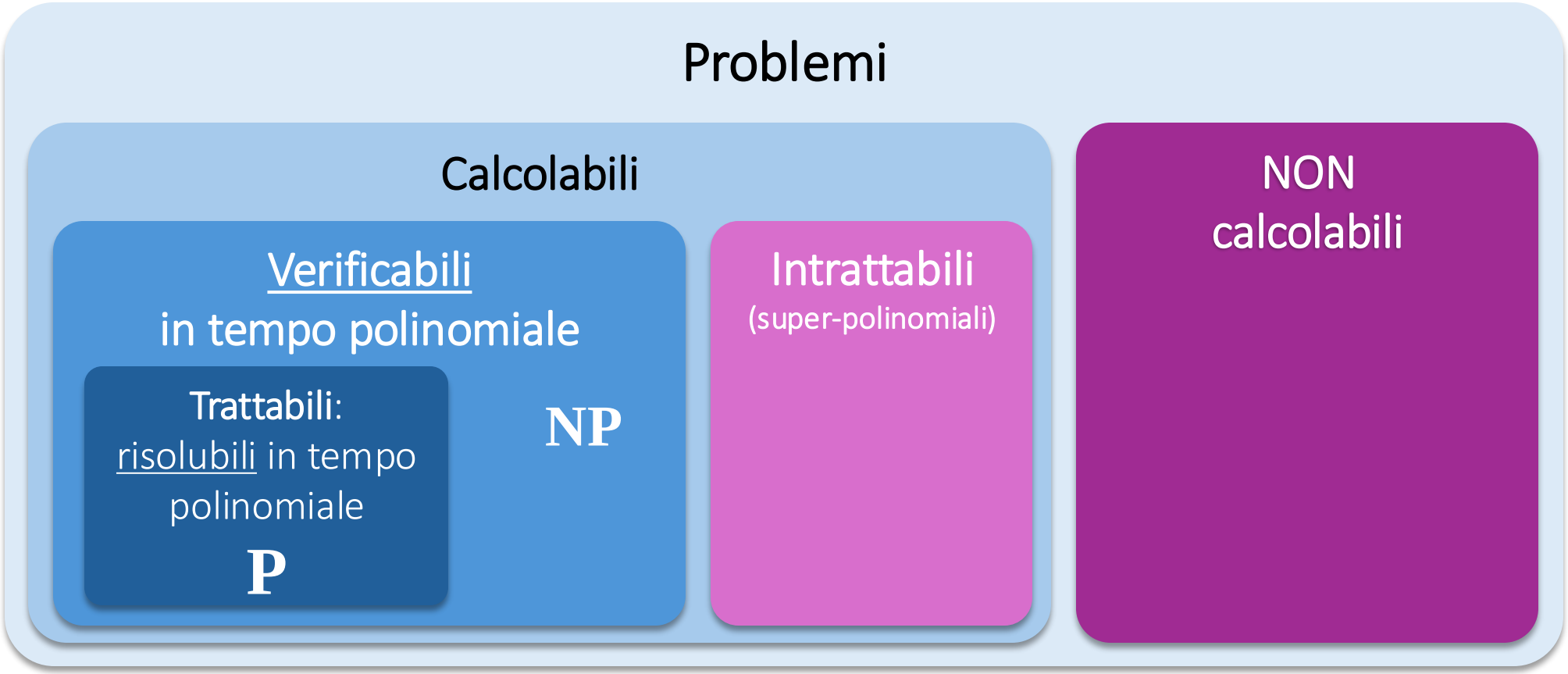
- Come sono fatti i problemi intrattabili? Sono complicatissimi?
- Somma di Sottoinsiemi: Dato un insieme di n numeri naturali A e un intero positivo k, esiste un sottoinsieme di A la cui somma degli elementi sia uguale a k?
- Partizione: Dato un insieme di n numeri naturali A, esiste una partizione di A in due sottoinsiemi X e Y tali che la somma degli elementi di X sia uguale alla somma degli elementi di Y?
- SAT: Data un'espressione logica booleana con n variabili booleane $x_1, \ldots, x_n$, esiste un'assegnazione di valori *vero/falso* alle variabili $x_1, \ldots, x_n$ tale da rendere vera l'intera espressione?

- L'unica strategia risolutiva nota, per questi problemi, è quella di provare tutte le possibili combinazioni: abbiamo in tutti e tre i casi 2^n possibili configurazioni da controllare

- Però se qualcuno mi suggerisse una soluzione per uno di questi problemi...
  in poco tempo potrei verificare se la soluzione è corretta oppure no

Problemi facilmente verificabili

Definiamo una nuova classe di problemi: quelli che ammettono un algoritmo di complessità polinomiale (quindi un algoritmo veloce, efficiente) per **verificare** se una soluzione è corretta oppure no

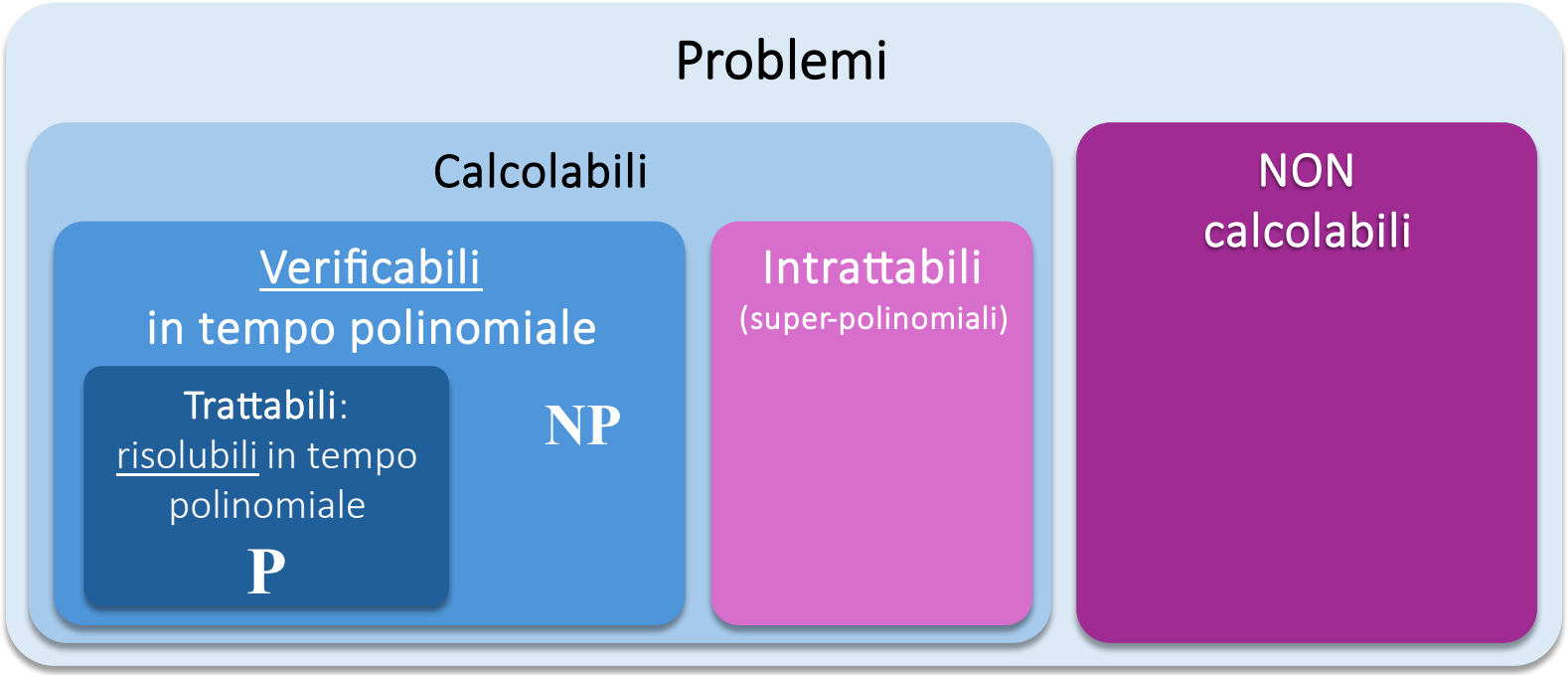


Il più grande problema dell'informatica

- Ma esistono veramente dei problemi che possono essere calcolati in tempo polinomiale da un modello di calcolo non deterministico e che invece richiedono un tempo esponenziale se calcolati mediante modelli di calcolo deterministici?

- Ossia: «**P = NP**»?

- Questo è il più grande problema aperto dell'informatica teorica: sulla sua soluzione il Clay Institute ha messo una taglia da $1.000.000!




Grazie per
l’attenzione!