ROMA TRE UNIVERSITÀ DEGLI STUDI

Dipartimento di SCIENZE DELLA FORMAZIONE

Seminario di Matematiche complementari
Federigo Enriques
Storia, didattica, epistemologia

SCIENZE
Dipartimento di Eccellenza

Calcolabilità

Marco Liverani

marco.liverani@uniroma3.it

http://www.mat.uniroma3.it/users/liverani/

Università degli Studi Roma Tre

Dipartimento di Scienze – Aula C

Giovedì 14 marzo 2024

Semplificare può essere «scivoloso»...

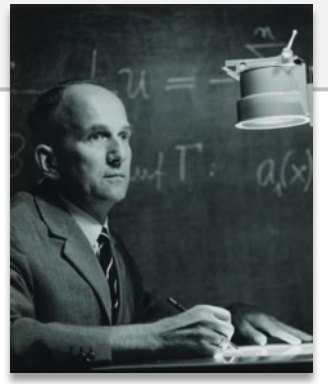
«Ogni tanto è necessario dire delle cose difficili,
ma bisognerebbe dirle nel modo più semplice di cui si è capaci.»

– Godfrey H. Hardy

Per cominciare... arriviamo dritti al punto

Esistono problemi non calcolabili? Cosa significa «calcolabilità»?

Problema



Lothar Collatz
(1910 – 1990)

- Dato un numero naturale $n > 0$:
  - se è pari lo si divida per 2
  - se è dispari lo si moltiplichi per 3 e si sommi 1
  - dopo un numero finito di iterazioni di questo procedimento si ottiene 1
- Esempio:

$$n = 17 \to 17 \times 3 + 1 = 52 \to 52/2 = 26 \to 26/2 = 13 \to 13 \times 3 + 1 = 40 \to 40/2 = 20 \to$$
$$\to 20/2 = 10 \to 10/2 = 5 \to 5 \times 3 + 1 = 16 \to 16/2 = 8 \to 8/2 = 4 \to 4/2 = 2 \to 2/2 = 1$$

Stanislaw Ulam
(1909 – 1984)

- Questa procedura è nota come **Congettura di Collatz** (o di Ulam), ma è, appunto, una congettura:
  - Siamo sicuri che prima o poi terminerà per ciascun valore di n?
  - Esiste un numero per cui la procedura non termina mai?

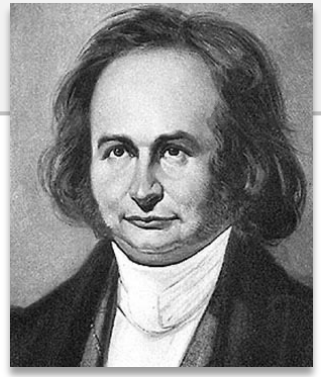
Problema

- Consideriamo il seguente problema: **calcolare un controesempio alla Congettura di Collatz/Ulam, ossia trovare un numero naturale n tale che, iterando la procedura di Collatz/Ulam, non si arrivi a 1**

- È facile scrivere un programmino per il computer per cercare la soluzione del problema: ingenuamente l'ho fatto quando a scuola, tanti anni fa, quando mi raccontarono qualcosa a proposito di questa strana procedura
- Il problema è che si tratta, appunto, di una ***congettura***: nessuno ha ancora mai dimostrato che sia vera!

- Se la Congettura di Collatz/Ulam dovesse essere vera il mio programmino non smetterà mai di girare (resteremo con il dubbio che tale numero esista); se dovesse essere falsa, invece, il programma non terminerà mai ugualmente, proprio perché non si arriva mai ad 1
- Dunque è possibile calcolare automaticamente la soluzione del problema? Non lo sappiamo!

Un altro problema



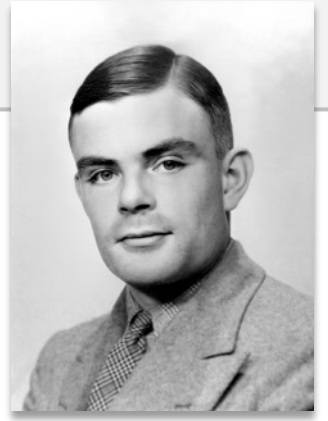
Christian Goldbach
(1690 – 1764)

- Ogni numero pari maggiore di 2 è dato dalla somma di due numeri primi
- Funziona per $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 5 + 5$, $12 = 7 + 5$, $94 = 41 + 53$, ...
- Funziona sempre? E chi lo sa?
- Si tratta della celeberrima Congettura di Goldbach, un problema di teoria dei numeri formulato nel 1742 in una lettera che Christian Goldbach scrisse a Eulero (il più grande matematico del tempo)

Leonhard Euler
(1707 – 1783)

- Problema: **calcolare un numero pari maggiore di 2 che non possa essere espresso come somma di due numeri primi**
- Potremmo scrivere una procedura sul computer per calcolare la soluzione del problema, ma purtroppo ... ci risiamo! Se la congettura di Goldbach fosse vera, il programma non terminerà mai, se invece fosse falsa, troverebbe un controesempio che smentisce la congettura
- Dunque la soluzione del problema è calcolabile automaticamente? Non lo sappiamo!

Problema della fermata



Alan Turing
(1912 – 1954)

- È possibile definire una **Macchina di Turing** M che sia in grado di stabilire se, data una Macchina di Turing M' e un input x per tale macchina, M' termini dopo un numero finito di passi elaborando x?
- In altri termini: è possibile scrivere un **algoritmo** A (o un programma per il computer) che preso in input un algoritmo A' (o un programma per il computer) e un input x, stabilisca se A' termina dopo un numero finito di passi elaborando x?
- La risposta, che rese celebre Alan Turing nel 1936, è **NO**: non è possibile definire una macchina di Turing (o un algoritmo o un programma per un computer «seriale») che stabilisca se la macchina di Turing M' per un input x termina la sua esecuzione dopo un numero finito di passi
- In conclusione: **esistono problemi non calcolabili**!
- Esistono problemi che la cui soluzione non può essere calcolata automaticamente, neanche con il computer più potente del mondo
- Questo risultato è talmente sconcertante che potremmo anche finire qui il nostro seminario

Calcolabilità

- Occuparsi di calcolabilità significa stabilire se un problema sia o meno calcolabile, ossia se la sua soluzione possa essere calcolata con un procedimento deterministico in un tempo finito, effettuando un numero finito di operazioni elementari

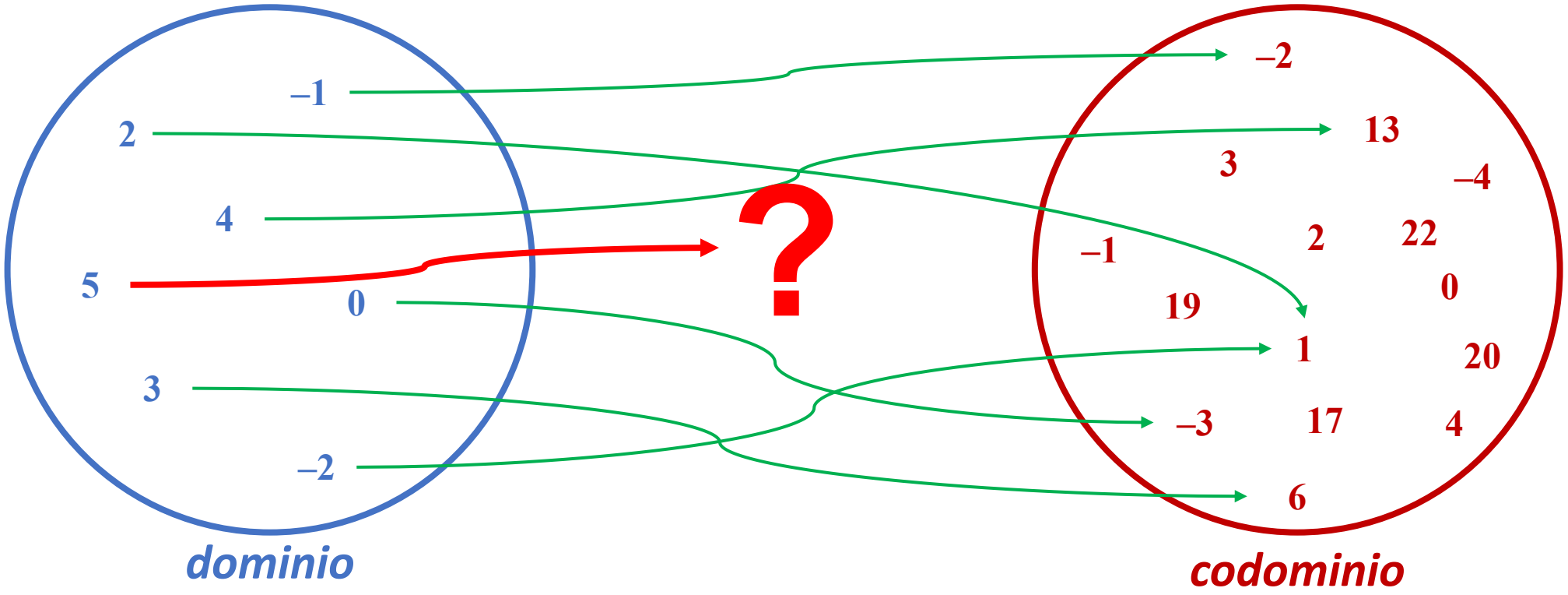
- Scoprire che non tutti i problemi siano calcolabili, apre degli scenari molto interessanti, ponendo dei limiti invalicabili al calcolo automatico/algoritmico (e quindi, generalizzando, al calcolo)

Ricominciamo da capo

Facciamo chiarezza, costruiamo un percorso per ragionare sulla calcolabilità

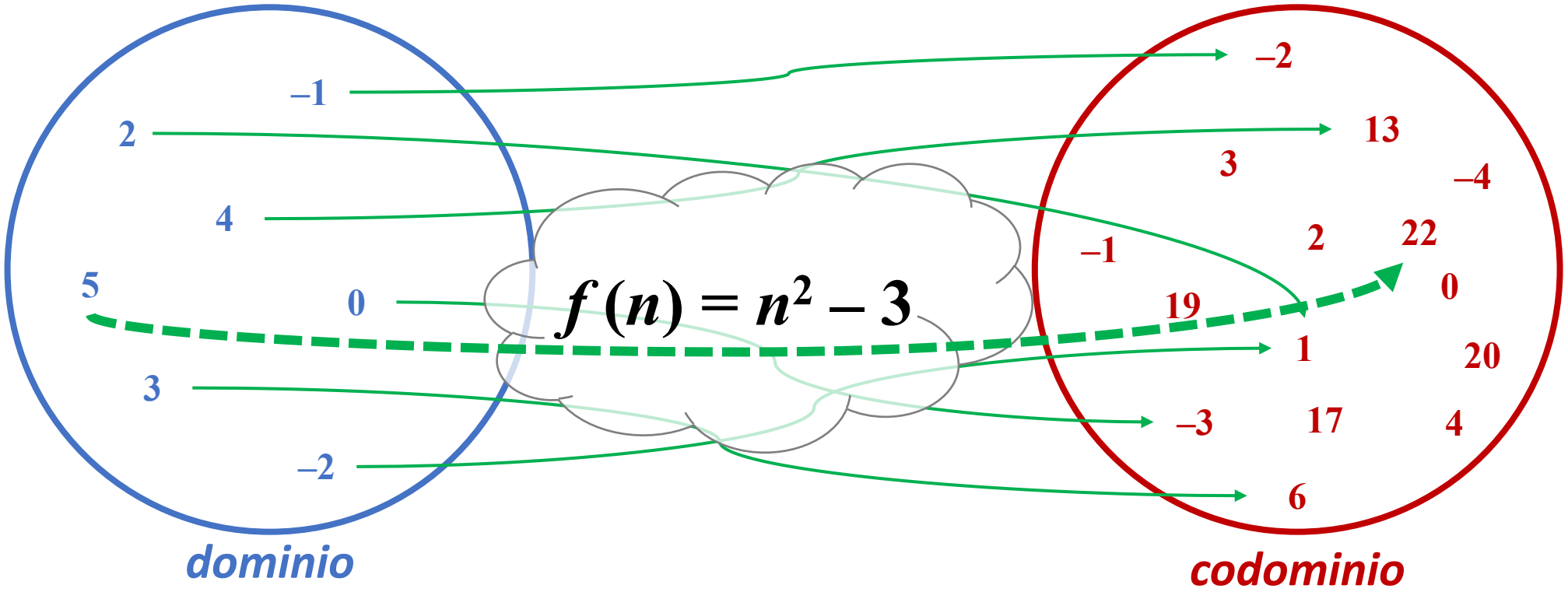
Funzioni

- 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 con 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

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, ...)
  - 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 dalle scuole elementari 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 complesse, 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)» = «3 2 12 4 – 2 / + ×»

3 2 **12 4 –** 2 / + × → 3 2 **8 2 /** + × → 3 **2 4 +** × → **3 6 ×** → **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 e prima di loro 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 il valore di x tale che $3x \times 5 + 4x \times 2 = 46 \Rightarrow x = 2$

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. $y := x \times x$
2. $y := y - 3 \times x$
3. $y := y + 5$
4. restituisci y

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. $s := s + k$
4. $k := k + 1$
5. se $k \leq n$ vai al passo 3 altrimenti prosegui
6. restituisci s

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 (*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 $n = 9$:

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

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 due funzioni calcolabili, allora anche $h = g \circ f$ è calcolabile
  - la ***ricorsione*** (le strutture algoritmiche iterative): $f(0) = k, \quad f(n) = g(n, f(n-1))$

- Per occuparci di **calcolabilità**, ossia di stabilire ciò che può essere calcolato in modo effettivo con un algoritmo, occorre formalizzare meglio il concetto di **algoritmo**

Modelli di calcolo

- Un modello di calcolo è un modello astratto con cui sono formalizzati gli aspetti sintattici e semantici del «linguaggio» che utilizzeremo per definire gli algoritmi, le procedure per la risoluzione automatica di un problema
- Basarsi su un modello di macchina astratta è necessario per poter confrontare l'**efficienza** di due differenti *algoritmi*, a prescindere dalla loro implementazione e dalla velocità del computer su cui eseguiremo il programma codificato in linguaggio macchina
- Basandoci su un modello di calcolo **astratto**, ma **semplificato**, è più semplice dimostrare in termini rigorosi la *correttezza* di un determinato algoritmo
- Definire con precisione il modello di calcolo a cui facciamo riferimento è fondamentale per costruire una teoria della calcolabilità (che altrimenti sarebbe lasciata solo all'intuito e all'immaginazione)

La Macchina di Turing

Alan Turing
(1912 – 1954)

- Ideata nel 1936 dal matematico inglese Alan Turing, una delle figure più importanti tra quelle che hanno contribuito alla definizione e allo sviluppo del calcolo automatico e della teoria Informatica
- È una macchina **astratta** basata su due componenti:
  - Un **nastro infinito** (da questa caratteristica ne segue il fatto che la macchina di Turing non è realizzabile praticamente) su cui la macchina può leggere e scrivere mediante
  - Una **testina di lettura e scrittura** che, scorrendo sul nastro, è in grado di leggerne e modificarne il contenuto
- La Macchina di Turing (MdT) è un'**estensione** di quello che gli studiosi di informatica teorica definiscono come un Automa a Stati Finiti Deterministico, con in più un nastro (che è al tempo stesso memoria, unità di input e di output) di lunghezza infinita

La Macchina di Turing

- Più formalmente una Macchina di Turing è una quintupla $\langle \mathcal{A}, \mathcal{S}, \delta, s_0, F \rangle$, dove:
  - $\mathcal{A}$ è un **alfabeto di simboli** che l'automa è in grado di elaborare (confrontare, leggere, scrivere)
  - $\mathcal{S}$ è l'**insieme degli stati** in cui si può trovare l'automa nel corso dell'elaborazione
  - δ è la **funzione di transizione** che fa passare l'automa da uno stato di $\mathcal{S}$ ad un altro sulla base del simbolo che sta elaborando e dello stato in cui si trova:

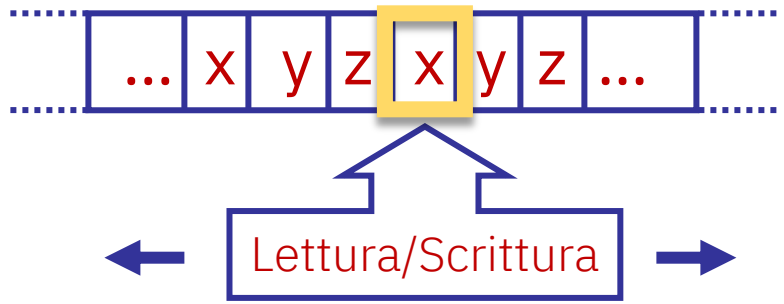
$$\delta : \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{A} \times \mathcal{S} \times \{avanti, indietro, ferma\}$$

  - $s_0 \in \mathcal{S}$ è lo **stato iniziale**, lo stato in cui si trova l'automa all'inizio dell'elaborazione
  - $F \subset \mathcal{S}$ è l'**insieme degli stati terminali**: quando l'automa raggiunge uno di questi stati l'elaborazione termina
- Gli stati terminali possono avere un significato diverso, a seconda del problema che si intende risolvere con l'automa (es.: successo, insuccesso, ecc.)

La Macchina di Turing

- Il funzionamento della macchina è basato sul cambiamento di stato sulla base del contenuto (del simbolo) che è presente sul nastro in corrispondenza della testina di lettura/scrittura
- Dopo aver letto il contenuto della posizione corrente del nastro, la macchina, sulla base dello stato in cui si trova, è in grado di passare in un altro stato, scrivere qualcosa nella posizione corrente del nastro ed infine spostarsi a destra o a sinistra sul nastro stesso
- Il nastro è infinito, mentre gli stati sono in quantità finita
- Il fatto che sia deterministico sta ad indicare che per ogni coppia "stato/simbolo" che viene incontrata dalla macchina, è **univocamente** determinata l'azione compiuta (scrittura e spostamento del nastro) e il nuovo stato in cui passa la macchina stessa; in questo modo la macchina può eseguire solo un'operazione per volta

La Macchina di Turing



- Per ogni problema che si intende risolvere è necessario progettare una macchina di Turing adeguata
- Per farlo è necessario definire:
  - L'**alfabeto** dei simboli che è possibile leggere e scrivere sul nastro
  - Gli **stati** in cui si può trovare la macchina
  - Le **transizioni** da uno stato ad un altro
  - Lo **stato iniziale** ed un insieme di **stati finali**
- Per rappresentare l'automa possiamo disegnare un **grafo degli stati**, ovvero una **matrice di transizione**

La Macchina di Turing: un esempio

- Esempio: letta una stringa di caratteri alfabetici, stabilire se termina o meno con la lettera «a»
- Alfabeto: a, b, c, …, x, y, z, # (il carattere «#» marcherà la fine della stringa)
- Strategia risolutiva:
  - Partendo dal primo carattere della stringa la macchina scorre il nastro verso destra fino a quando non incontro il carattere «#»
  - Scorrendo il nastro verso destra, ogni volta che la macchina trova un carattere «a» si pone in uno *stato di preallarme* (S_1) e torna nello *stato di quiete* (S_0) quando invece incontra una lettera dalla «b» alla «z»
  - Se quando si incontra il «#» la macchina si trova nello stato S_0 allora questo significa che la stringa non termina con la lettera «a» (e passa quindi nello stato S_2), altrimenti, se si trova in S_1, vuol dire che la stringa termina con la lettera «a» (e passa quindi nello stato S_3)

La Macchina di Turing: un esempio

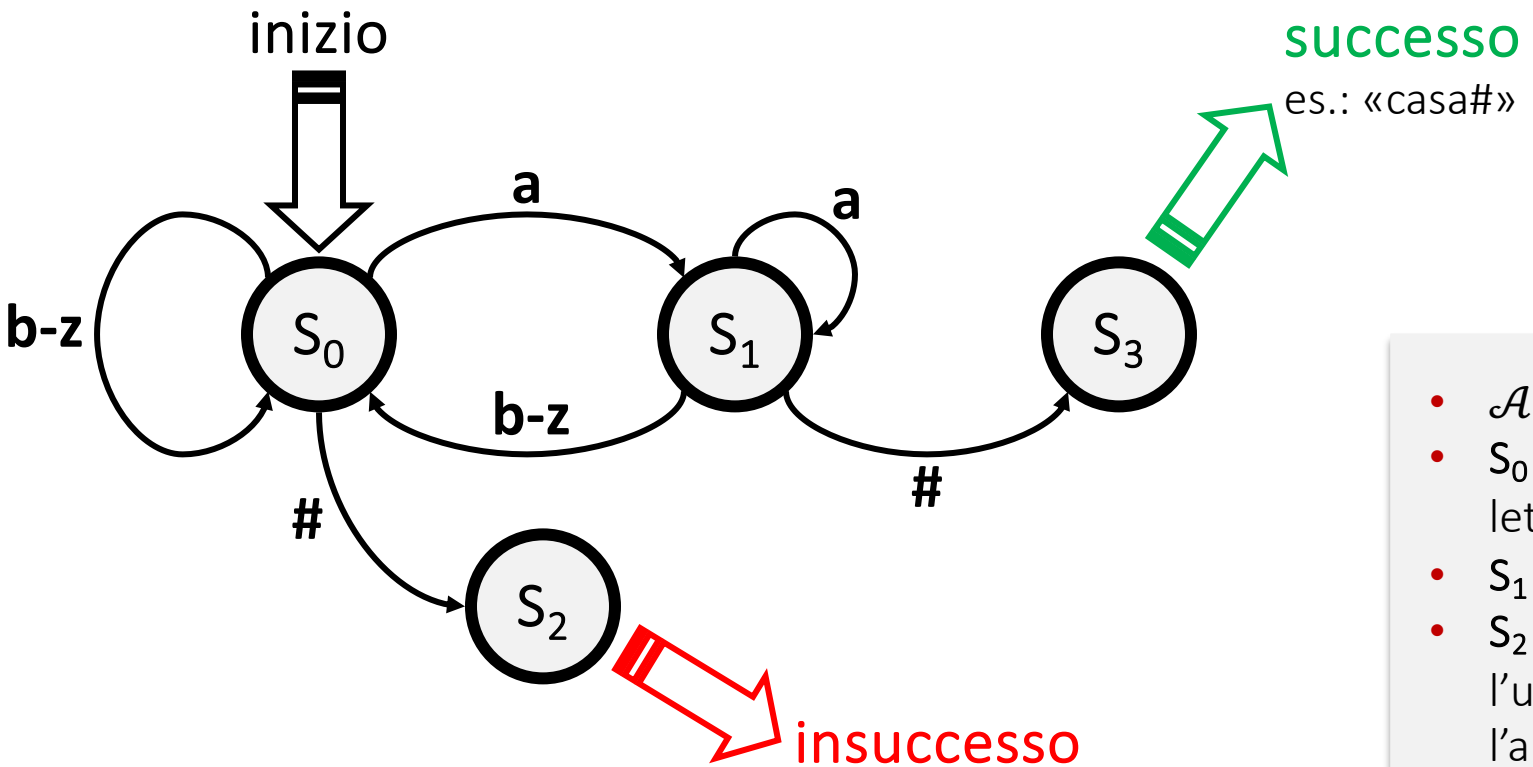
- È possibile descrivere una macchina di Turing attraverso una matrice di transizione; ad esempio:

	S_0	S_1	S_2	S_3
a	-, Dx, S_1	-, Dx, S_1	n.d.	n.d.
b-z	-, Dx, S_0	-, Dx, S_0	n.d.	n.d.
#	-, -, S_2	-, -, S_3	No	Sì

- Per ogni coppia carattere/stato la matrice indica l'azione che deve intraprendere la macchina (scrivere, spostarsi di una posizione sul nastro, passare ad un altro stato)

La Macchina di Turing: un esempio

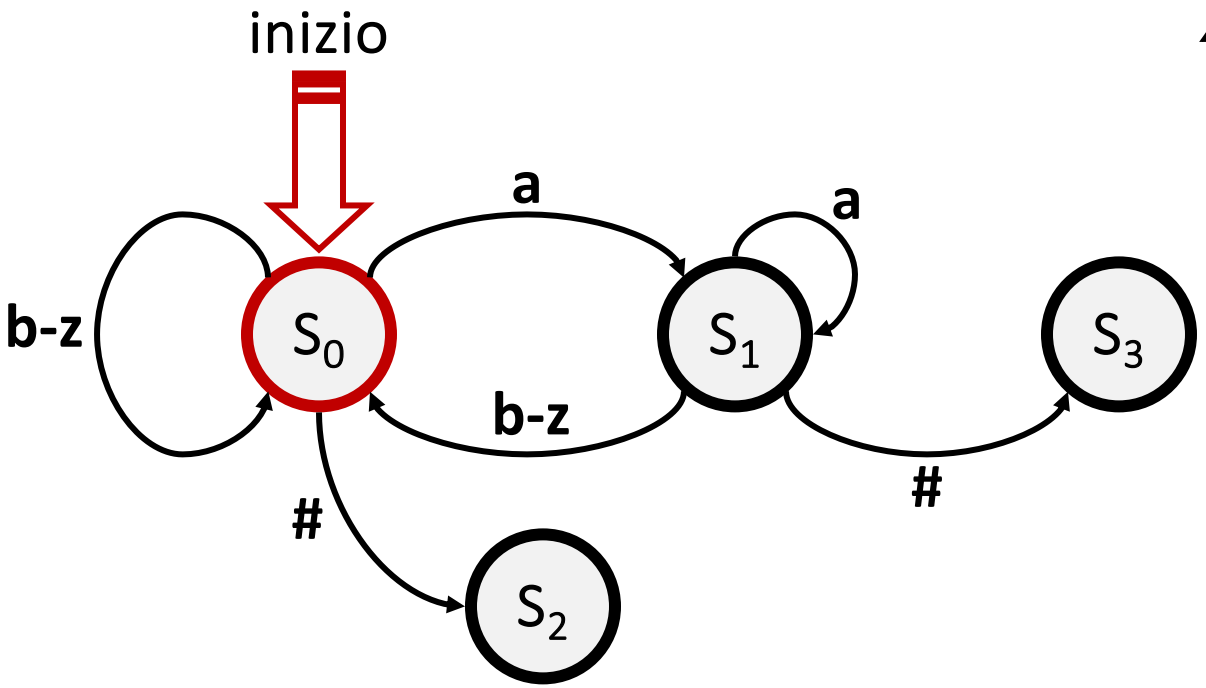
- Grafo delle transizioni di stato:

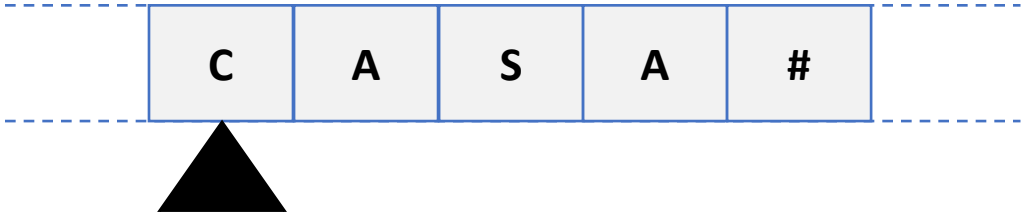


- $\mathcal{A}$ = {a, b, c, ..., x, y, z, #}
- S_0 = stato iniziale, oppure non è stata letta la lettera «a»
- S_1 = è stata letta la lettera «a»
- S_2 = la parola è finita (è stato letto «#») e l'ultima lettera non era la «a» perché l'automa era nello stato S_0
- S_3 = la parola è finita (è stato letto «#») e l'ultima lettera era la «a» perché l'automa si trovava nello stato S_1

La Macchina di Turing: un esempio

- Grafo delle transizioni di stato:

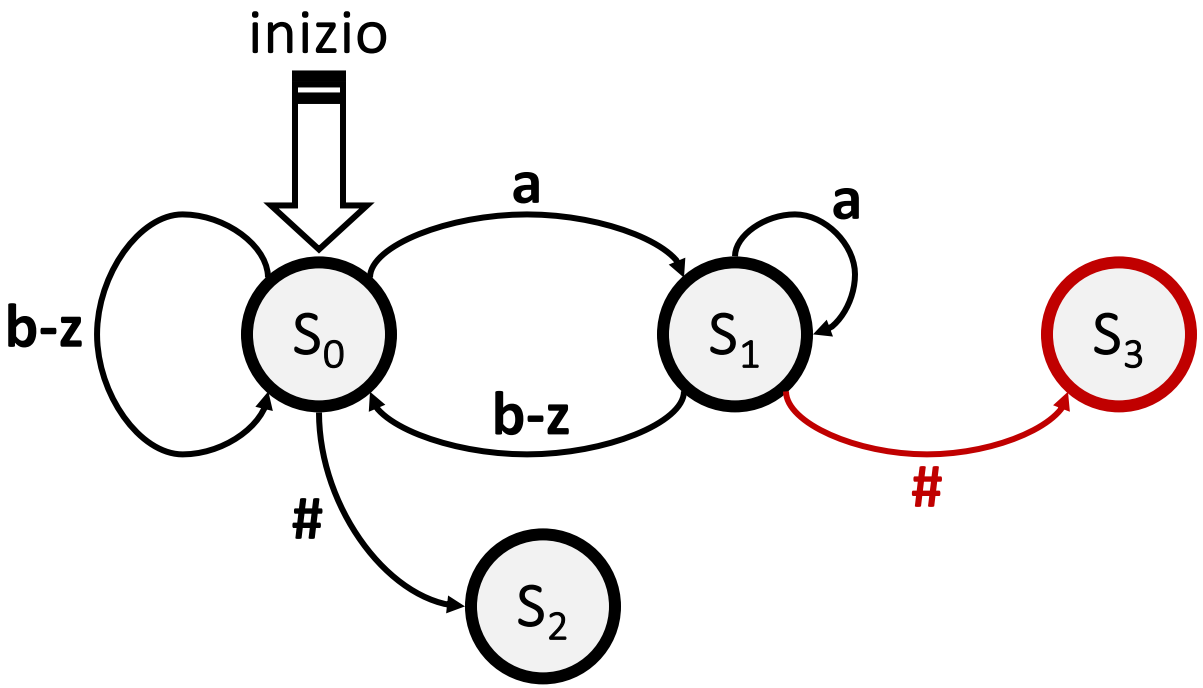


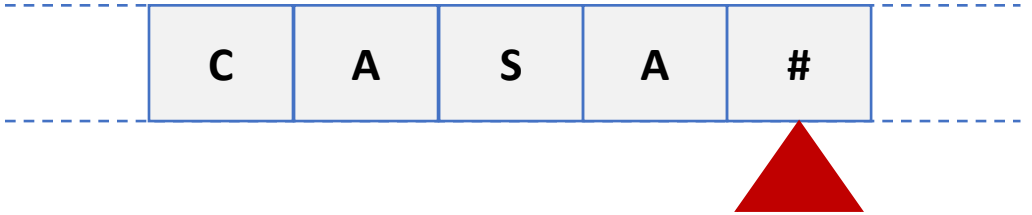


- $\mathcal{A}$ = {a, b, c, ..., x, y, z, #}
- S_0 = stato iniziale, oppure non è stata letta la lettera «a»
- S_1 = è stata letta la lettera «a»
- S_2 = la parola è finita (è stato letto «#») e l'ultima lettera non era la «a» perché l'automa era nello stato S_0
- S_3 = la parola è finita (è stato letto «#») e l'ultima lettera era la «a» perché l'automa si trovava nello stato S_1

La Macchina di Turing: un esempio

- Grafo delle transizioni di stato:

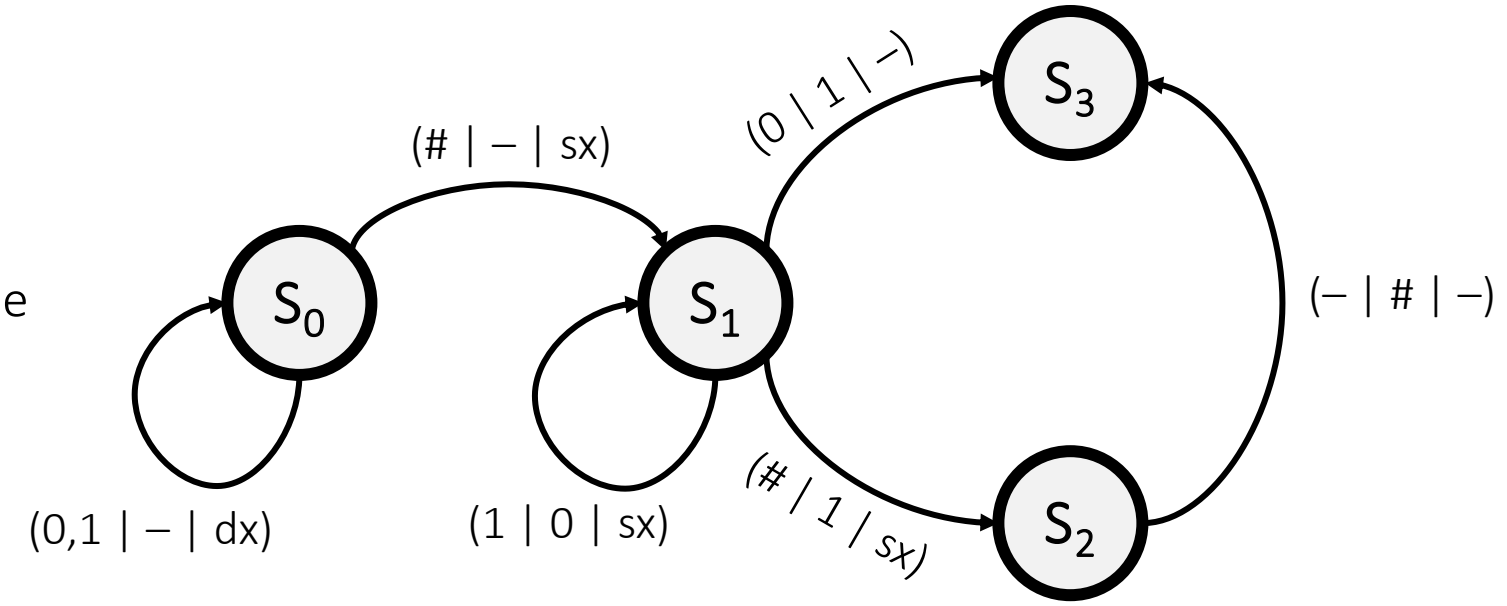




- $\mathcal{A}$ = {a, b, c, ..., x, y, z, #}
- S_0 = stato iniziale, oppure non è stata letta la lettera «a»
- S_1 = è stata letta la lettera «a»
- S_2 = la parola è finita (è stato letto «#») e l'ultima lettera non era la «a» perché l'automa era nello stato S_0
- S_3 = la parola è finita (è stato letto «#») e l'ultima lettera era la «a» perché l'automa si trovava nello stato S_1

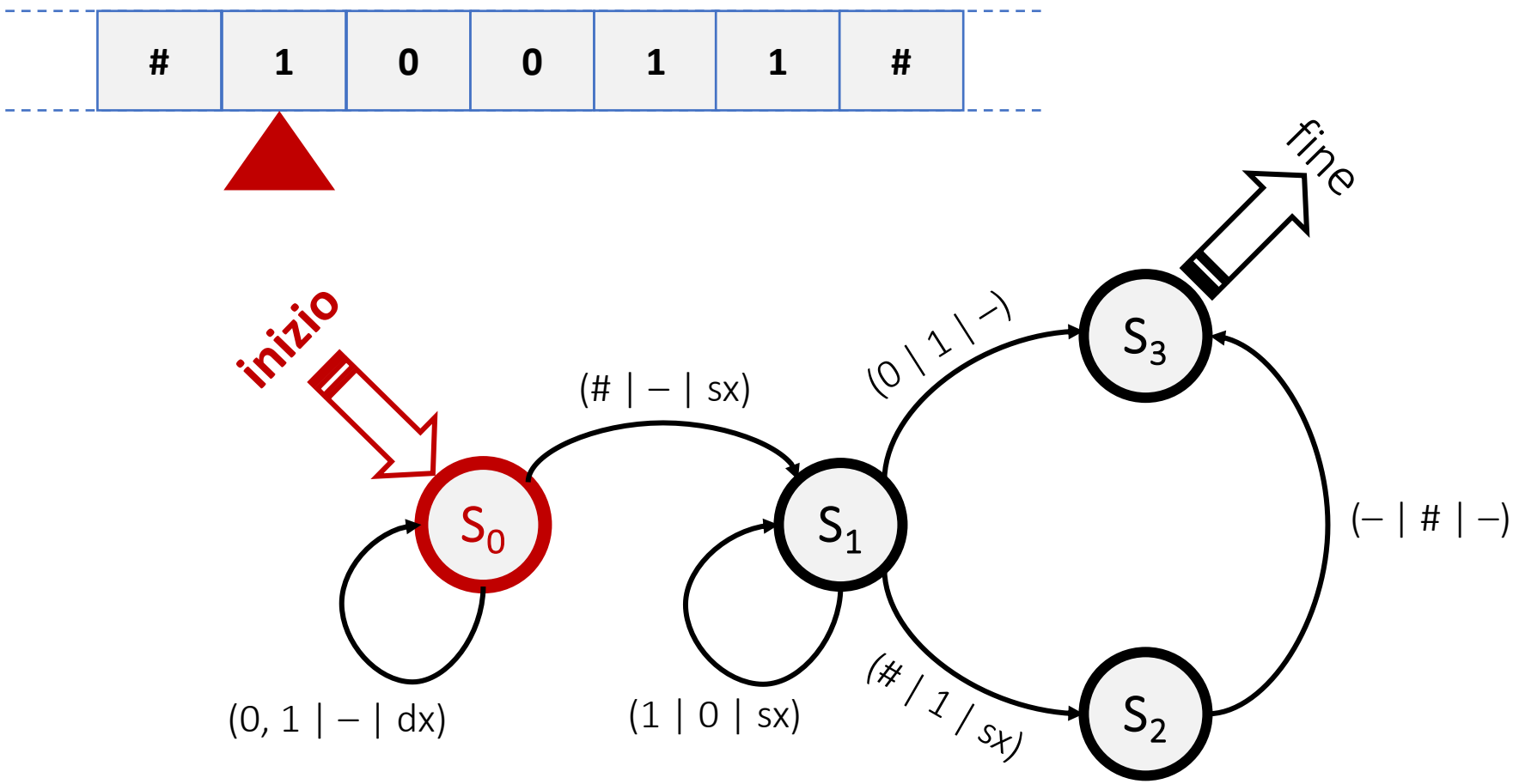
La Macchina di Turing: calcoli numerici

- È possibile mostrare come sia possibile definire Macchine di Turing per eseguire le operazioni aritmetiche di base, codificando le informazioni numeriche in modo opportuno (tipicamente si usala notazione «unaria» in base 1)
- Ad esempio vediamo come sia possibile definire la funzione «successore» (come negli Assiomi di Peano) che ad ogni numero naturale n fa corrispondere il numero naturale $n + 1$
- Usiamo una rappresentazione dei numeri in base 2 e definiamo una Macchina di Turing:
  - alfabeto $\mathcal{A}$ = {0, 1, #}
  - insieme degli stati $\mathcal{S}$ = {S_0, S_1, S_2, S_3}
  - lo stato iniziale $S_0 \in \mathcal{S}$
  - lo stato finale $S_3 \in \mathcal{S}$
  - esprimiamo la funzione di transizione di stato δ con il grafo



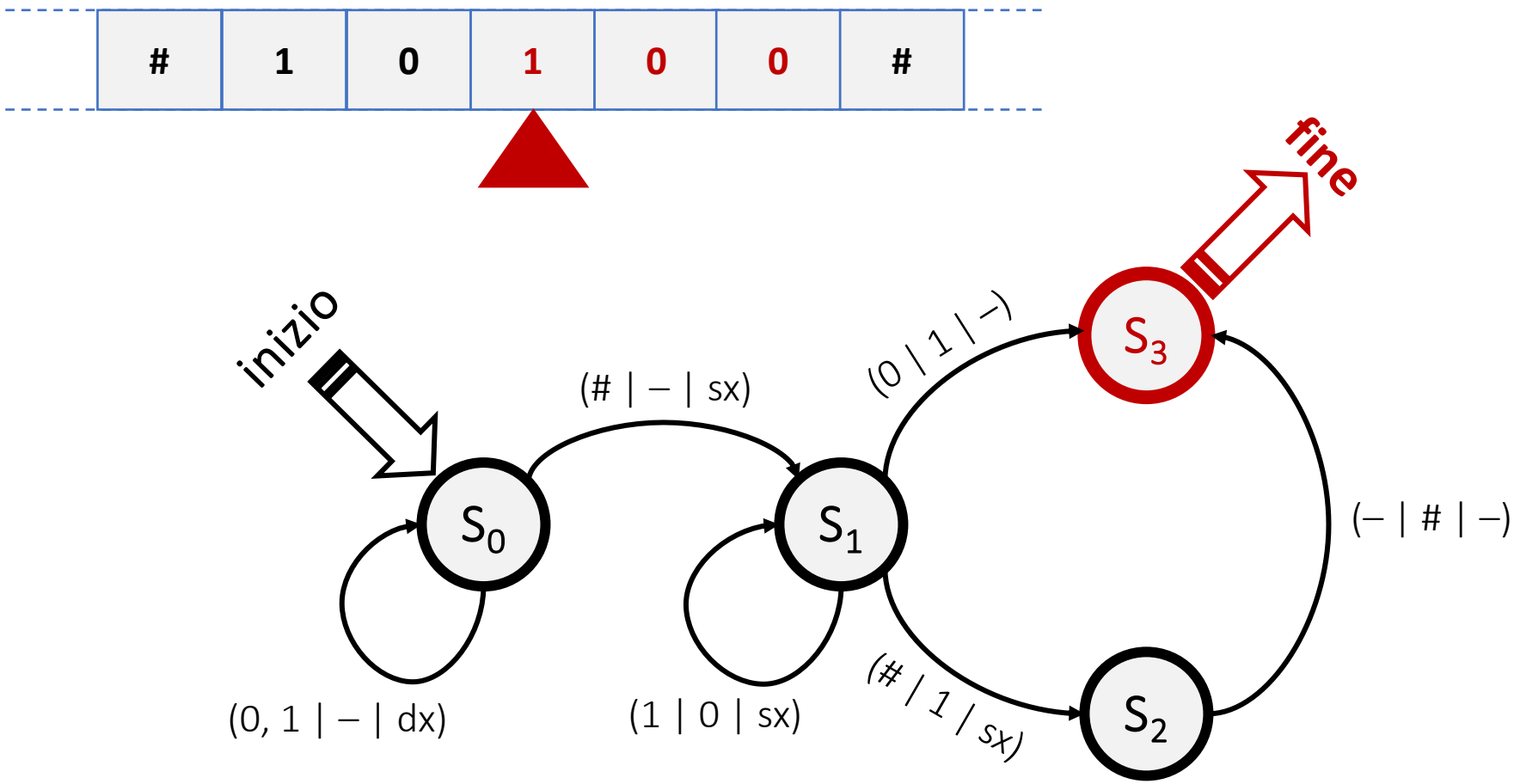
La Macchina di Turing: calcolo del successore

- Esempio: $19_{10} = 10011_2 \Rightarrow 19 + 1 = 20_{10} = 10100_2$



La Macchina di Turing: calcolo del successore

- Esempio: $19_{10} = 10011_2 \Rightarrow 19 + 1 = 20_{10} = 10100_2$



Codifica di una Macchina di Turing

- Una Macchina di Turing (MdT) possiamo quindi descriverla come un insieme di «istruzioni» definite attraverso una **quintupla** $\langle s_i , c , s_f, c', m \rangle$, dove s_i è lo stato in cui si trova la macchina e c è il carattere che legge sul nastro, s_f è lo stato in cui passa la macchina (può anche essere $s_f = s_i$), c' è il carattere che viene scritto sul nastro (può essere $c' = c$), m è lo spostamento sul nastro (può essere «*sinistra*», «*destra*», «*fermo*»)
- Siccome i valori che possono essere assunti da ciascun elemento di una quintupla (gli stati, i caratteri e gli spostamenti) sono di cardinalità finita, possiamo associare a ciascun valore un **numero naturale dispari** (es.: 1 = spostamento a sinistra, 3 = spostamento a destra, 5 = fermo, 7 = stato S_0, 9 = stato S_1, ...)
- Ogni istruzione/quintupla può essere così identificata da una **quintupla** di numeri naturali **dispari**: $\langle a, b, c, d, e \rangle$

Codifica di una Macchina di Turing

- Se $\langle a, b, c, d, e \rangle$ sono i cinque numeri che rappresentano la k-esima quintupla / istruzione della MdT, possiamo identificare tale quintupla/istruzione con un numero naturale **pari** costruito con l'espressione

$$I_k = 2^a \times 3^b \times 5^c \times 7^d \times 11^e$$

  usando come basi delle potenze **i primi cinque numeri primi**

- L'intera MdT è data quindi dalla sequenza (finita) di numeri $I_1, I_2, I_3, \ldots, I_n$ e possiamo quindi identificarla con il numero naturale

$$M = 2^{I_1} \times 3^{I_2} \times 5^{I_3} \times \cdots \times p_n^{I_n}$$

  dove $2, 3, 5, \ldots, p_n$ sono i primi n **numeri primi**

- I numeri I_k e M sono tutti diversi tra loro, anche perché gli I_k hanno esponenti dispari, mentre i numeri M hanno esponenti pari

La Macchina di Turing Universale

- Siccome ogni Macchina di Turing è quindi identificata univocamente da un determinato numero naturale M ottenuto mediante una fattorizzazione delle sue istruzioni, possiamo ipotizzare la definizione di una **Macchina di Turing Universale** che acquisisca in input
  - una Macchina di Turing M (rappresentata sul nastro da una opportuna codifica del numero naturale M che la rappresenta)
  - i dati su cui la MdT M deve operare
- La Macchina di Turing Universale «esegue» le istruzioni della MdT M e ne produce in output il risultato
- La Macchina di Turing Universale è quindi una MdT estremamente duttile, programmabile attraverso MdT

Problemi non calcolabili

David Hilbert
(1862 – 1943)

- Problema della decisione («Entscheidungsproblem», Hilbert & Ackermann 1928):

  è possibile definire un algoritmo che dato un determinato enunciato, stabilisca se è vero oppure no?

Wilhelm Ackermann
(1896 – 1962)

- Problema della fermata («halting problem»):

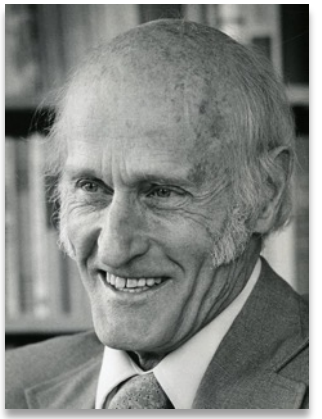
  è possibile definire una Macchina di Turing Universale che, data in input una Macchina di Turing X (o un algoritmo deterministico), verifichi se X termina dopo un numero finito di operazioni per qualsiasi istanza del problema?

- È il primo esempio di problema non calcolabile, ossia non è possibile definire una Macchina di Turing (o un algoritmo deterministico) che lo risolva; in altri termini non è un problema risolubile (calcolabile) attraverso una Macchina di Turing Universale

Modelli Turing-equivalenti

- La Macchina di Turing non è l'unico modello di calcolo astratto, ne sono stati definiti altri:
  - Lambda calcolo di Alonzo Church (1936)
  - Funzioni ricorsive di Kleene (1936)
  - Processi combinatori finiti di Post (1936)
  - i principali linguaggi di programmazione (C, Python, Java, ...)
- Sono modelli equivalenti (in termini di capacità di calcolo) alla Macchina di Turing: un problema è calcolabile con una Macchina di Turing se e solo se è calcolabile con un altro modello equivalente

Alonzo Church
(1903 – 1995)



Stephen Kleene
(1909 – 1994)

Tesi di Church-Turing

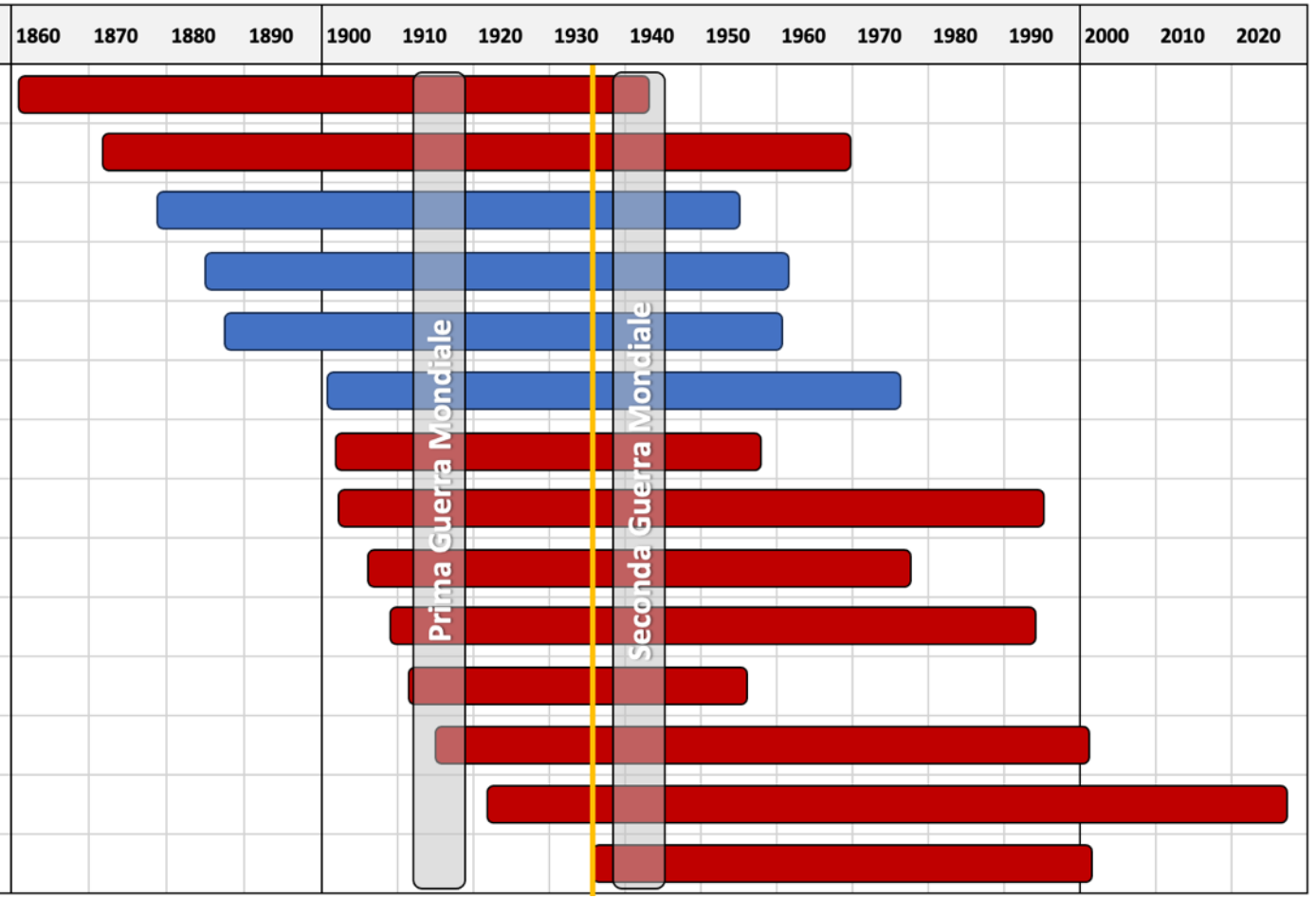
Tesi di Church-Turing: *una funzione è calcolabile se e solo se è Turing-calcolabile*

- È una tesi non dimostrata, ma Kleene propone quattro argomenti a sostegno della sua veridicità:
  1. **Non confutazione**: nonostante i numerosi tentativi, la Tesi di Church-Turing non è mai stata confutata
  2. **Confluenza**: ben quattro modelli equivalenti furono proposti contemporaneamente e tutti conducono definire la stessa classe di funzioni calcolabili
  3. **Analisi di Turing**: le capacità di calcolo «umano» coincidono con quelle di una MdT o di un modello equivalente
  4. **Logica del primo ordine**: il calcolo è una forma di deduzione matematica e quindi può essere formalizzato come una deduzione valida nel linguaggio della logica dei predicati del primo ordine con identità e può quindi essere eseguito dalla macchina di Turing universale

Evoluzione del pensiero scientifico nel ‘900

- Lo sviluppo della teoria della calcolabilità e dell’informatica è andata di pari passo con lo sviluppo della matematica, della fisica quantistica e della fisica nucleare, contaminandosi e supportandosi vicendevolmente

Nome	Nazionalità	Nascita	Morte	Principali contributi
David Hilbert	Germania	1862	1943	Matematica
Bertrand Russel	Gran Bretagna	1872	1970	Logica matematica
Albert Einstein	Germania / Svizzera / Stati Uniti	1879	1955	Fisica, Teoria della relatività
Niels Bohr	Danimarca	1885	1962	Fisica quantistica
Erwin Schrödinger	Austria	1887	1961	Fisica quantistica
Werner Heisenberg	Germania	1901	1976	Fisica quantistica, Principio di indeterminazione
John Von Neumann	Ungheria / Stati Uniti	1903	1957	Teoria dei giochi, Fisica nucleare, Informatica, ...
Alonzo Church	Stati Uniti	1903	1995	Logica, calcolabilità, lambda calcolo
Kurt Gödel	Austria / Stati Uniti	1906	1978	Logica matematica, Teoremi di incompletezza
Stephen Kleene	Stati Uniti	1909	1994	Logica, calcolabilità, funzioni ricorsive
Alan Turing	Gran Bretagna	1912	1954	Logica, calcolabilità, crittografia, informatica
Claude Shannon	Stati Uniti	1916	2001	Teoria dell'informazione, teoria dei codici, ...
Corrado Böhm	Italia	1923	2017	Logica, calcolabilità, informatica
Giuseppe Jacopini	Italia	1936	2001	Logica, calcolabilità, informatica



Complessità

Possiamo effettivamente calcolare la soluzione di tutti i problemi 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 d'acqua per la pasta 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 calcolabile, ma la procedura per risolverlo è impraticabile**

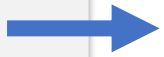
Complessità di un algoritmo

- 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

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



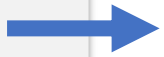
questa operazione viene eseguita $(n-1) + (n-2) + ... + 1$ volte

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



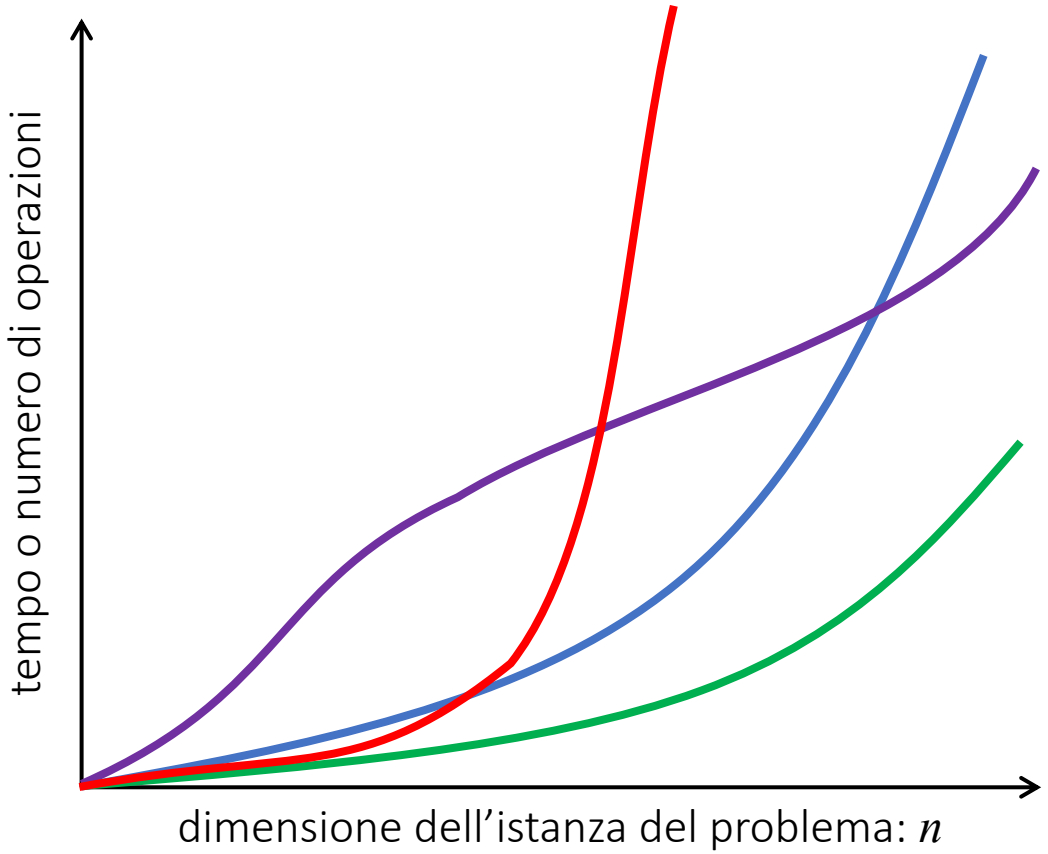
questa operazione viene eseguita $(n-1) + (n-2) + \ldots + 1$ volte

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

Classi di funzioni di complessità

- Indicheremo con $O(f(n))$ l'insieme delle funzioni che hanno un andamento asintotico dominato da $f(n)$, a meno di costanti:

$$O(f(n)) = \{g(n) \text{ tali che esistono } c > 0 \text{ e } n_0 > 0 : cf(n) > g(n) \text{ per } n > n_0\}$$



$f(n)$ es.: $f(n) = n^2$

$g_1(n)$ es.: $g_1(n) = n^2/3$

$g_2(n)$ es.: $g_2(n) = \frac{1}{2}\, n^2 + 2n$

$$g_1(n), g_2(n) \in O(f(n))$$

$g_3(n)$ es.: $g_3(n) = n^3$

$$g_3(n) \notin O(f(n))$$

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 esegue **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

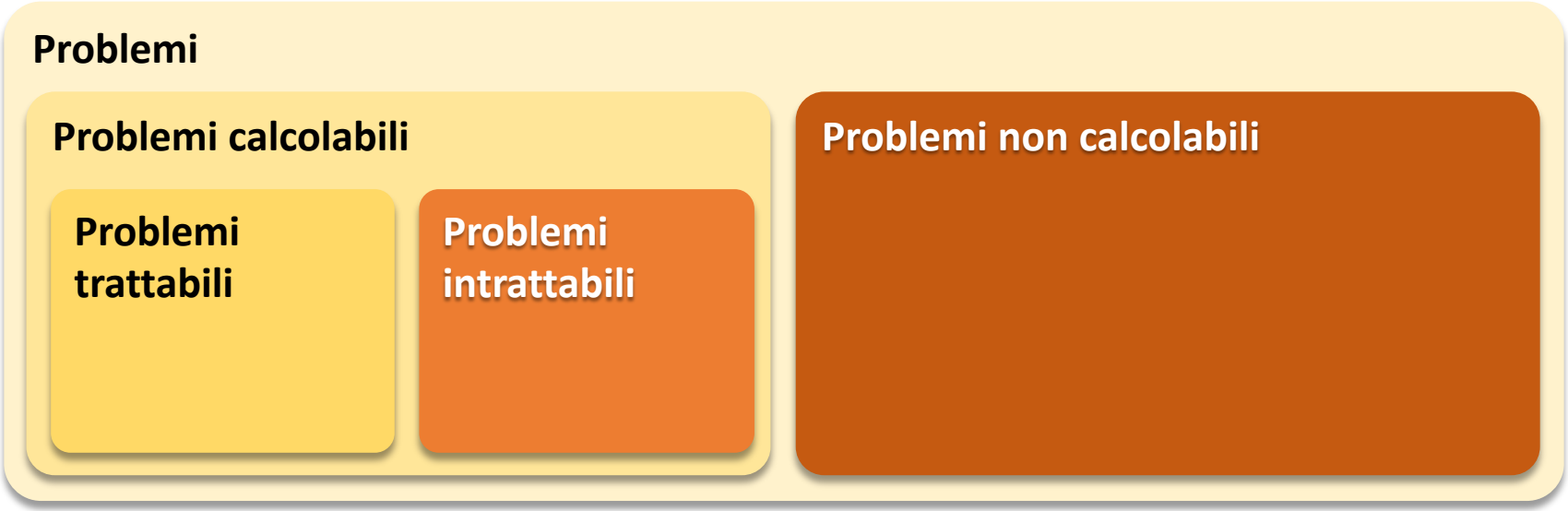
L'ultimo computer Apple con CPU «M3 MAX» immesso sul mercato esegue **18.000 miliardi** di operazioni al secondo

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 ($\mathbf{P}$)



Intrattabile = difficilissimo?

- Come sono fatti i problemi intrattabili? Sono complicatissimi?
- SUBSETSUM: 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?
- PARTITION: Dato un insieme di n numeri naturali A, esiste una partizione di A in due sottoinsiemi A_1 e A_2 tali che la somma degli elementi di A_1 sia uguale alla somma degli elementi di A_2?
- 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

La classe dei problemi NP

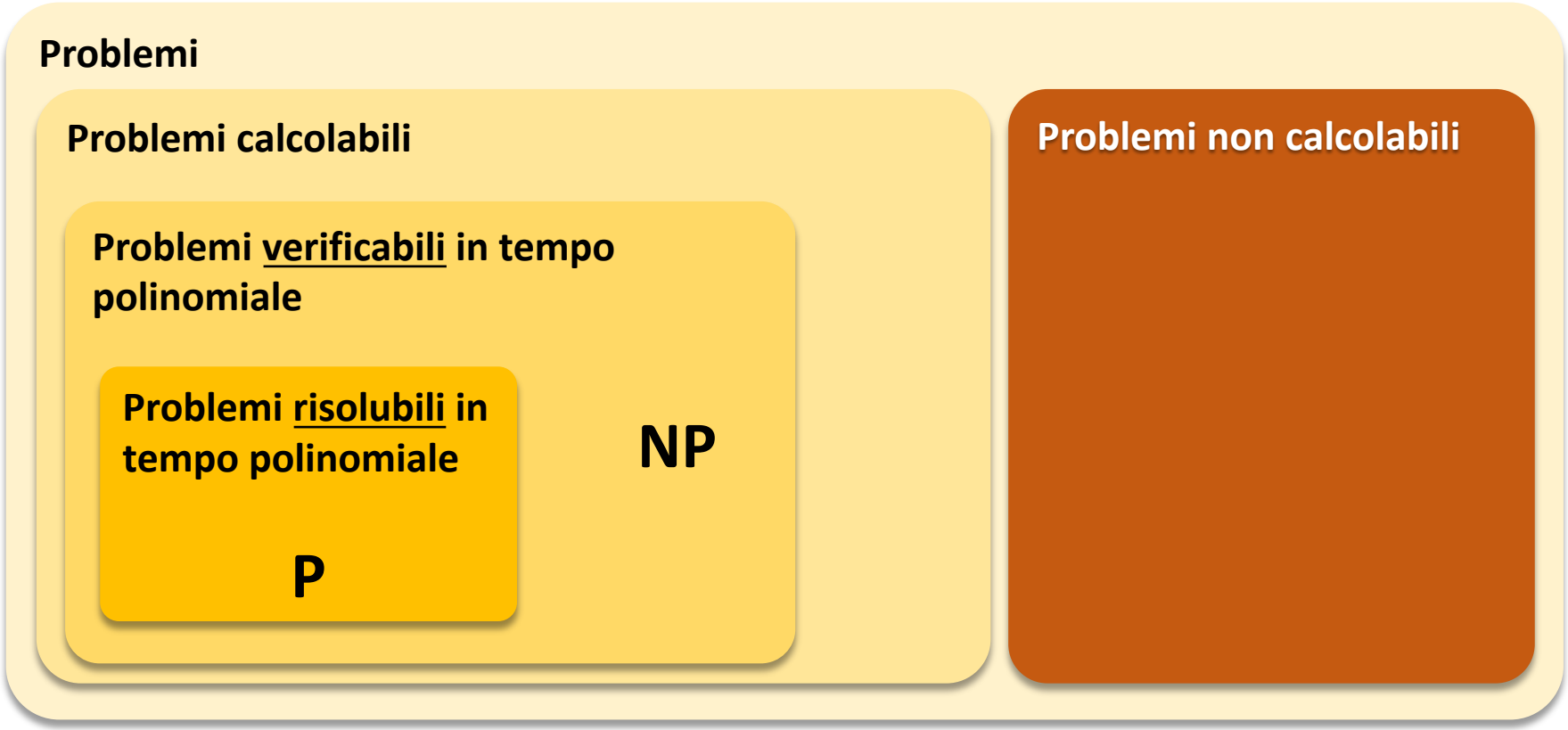
- Indichiamo con **NP** l'insieme dei **problemi che possono essere risolti in tempo polinomiale** da un algoritmo/macchina «**non deterministica**» (NP = *Non-deterministic Pynomial time*)
- Una macchina non deterministica è un modello di calcolo in grado di utilizzare contemporaneamente un numero arbitrario 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

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 su 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!

Calcolabilità e didattica

Proviamo a tracciare un «fil rouge» con la didattica della matematica

Forma e sostanza

- Il discorso che ho portato avanti fin qui ha una doppia chiave di lettura

    1. la prima è **diretta**, riguarda i temi della calcolabilità e della complessità computazionale di cui abbiamo parlato lungamente (seppur in modo fin troppo sintetico e superficiale)

    2. la seconda è **indiretta** e riguarda la formalizzazione e il linguaggio: questo aspetto mi sembra interessante anche dal punto di vista didattico

- Ho cercato di mettere in luce un aspetto assai rilevante della matematica e dell'informatica teorica che è quello relativo alla necessità di disporre di un **linguaggio** e di un **formalismo** adeguato per poter parlare di determinati argomenti

- Nel nostro caso, abbiamo avuto bisogno di definire il concetto di Macchina di Turing per poter circoscrivere un modello di calcolo e un'idea di algoritmo utile a discutere di calcolabilità
- Ciò che potrebbe apparire solo un aspetto formale, è invece un fatto sostanziale

Algoritmi e matematica

- L'uso degli algoritmi (con o senza la traduzione con un linguaggio di programmazione) può essere un utile esempio (o un esercizio) per aiutare i ragazzi a comprendere il linguaggio dell'aritmetica
- Raccontare un'espressione complicata, attraverso una successione (o una reiterazione) di passaggi elementari, può essere un efficace strumento di chiarezza e di verifica
- Spesso in matematica usiamo espressioni del tipo:

$$s = \sum_{k=1}^{n} a_k \qquad p = \prod_{k=1}^{n} a_k \qquad m = \min_{k=1,\ldots,n} a_k$$

  espressioni che possono essere chiarite con una procedura algoritmica iterativa
- La generalizzazione di una procedura di calcolo, attraverso l'uso del concetto di «*input*» e di «*output*» può essere utile per creare un parallelo con il concetto di funzione come corrispondenza tra due insiemi: il dominio (input) e il codominio (output)

Attenzione alle differenze

- L'uso di un linguaggio di programmazione mette però in luce anche diverse differenze che una sintassi apparentemente simile (tra linguaggio dell'aritmetica e degli algoritmi) può nascondere

- $a = 3$ è un'operazione di assegnazione di un valore ad una variabile, non è un'equazione

  altrimenti $a = a + 1$ non avrebbe senso in informatica

- $x^2 - 7 = 0$ in termini algoritmici non ha molto senso, piuttosto si può scrivere $x = \sqrt{7}$

- Anche le espressioni logiche nascondono delle differenze:

  «$x \neq 0$ and $y/x = 15$» non è equivalente a «$y/x = 15$ and $x \neq 0$»

Strumenti «poveri» ma stimolanti

- L'uso delle Macchine di Turing per formalizzare un procedimento di calcolo, sono uno strumento assai stimolante, proprio per povertà degli strumenti a disposizione di chi deve progettare una strategia risolutiva, un procedimento di calcolo
  - trovare una codifica «furba» delle informazioni
  - trovare una strategia risolutiva «innaturale», super-semplice, ma efficace
  - ragionare con uno strumento certamente completamente sconosciuto a tutti, mette tutti sulla stessa linea di partenza
  - ragionare alla progettazione di uno strumento astratto ha anche degli aspetti ludici e che possono creare una «competizione virtuosa» tra gruppi di studenti

Bibliografia essenziale

- Marcello Frixione, Dario Palladino, «*Funzioni, Macchine, Algoritmi - Introduzione alla teoria della computabilità*», Carocci, 2004
- Marcello Frixione, Dario Palladino, «*La computabilità: algoritmi, logica, calcolatori*», Carocci, 2017
- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, «*Automi, linguaggi e calcolabilità*», Addison-Wesley, 2003
- Marco Liverani, «*Qual è il problema – Metodi, strategie risolutive, algoritmi*», Mimesis, 2005


The End