

Alan M. Turing, 1912-1954



Programmazione 1

CdL Matematica

a.a. 2024-2025

Vincenzo Marra

Università degli Studi di Milano

Dipartimento di Matematica *Federigo Enriques*

24.II.2025

Logistica

- Docente. V. Marra, Studio in Area Sottotetto.
- Email. vincenzo.marra@unimi.it
- Ricevimento. Su appuntamento concordato per email o a lezione.
- Struttura del corso.
 - Lezioni frontali: in Aula Chisini. Turno unico: V.M.
 - Laboratorio: al Settore Didattico (via Celoria). Docenti:
 - Michele Barbato
 - Matteo Bianchi
 - Marco Bressan
 - Angelo Genovese
 - Andrea Visconti

Logistica

- Quattro turni di laboratorio, vedi Portale degli Orari.
 - Frontale
 - Il lunedì, 13:30-16:30, Aula Chisini
 - Quarti d'ora accademici: inizio lezioni 13:45; termine 16:15.
 - Laboratorio
 - Il mercoledì, 13:30 –16:30, Aule 307 e 309
 - Il giovedì, 13:30–16:30, Aule 307 e 309
- Pagina web di riferimento: sito MyAriel del corso.

Esami

- Non ci sono prove intermedie.
- Appelli ordinari:
 - Una prova in laboratorio di 3 ore.
 - Per chi supera la prova di laboratorio, un esame orale.

Nota: A certe condizioni è possibile verbalizzare il voto della prova di laboratorio senza sostenere la prova orale.

Bibliografia e Materiali

- **Diapositive delle lezioni e codice degli esempi.**
Pubblicati di volta in volta sul sito MyAriel.
- **Esercitazioni di Laboratorio.** Pubblicate di volta in volta sul sito MyAriel.
- **Testi di riferimento.** Consultate la sezione apposita del sito MyAriel (in aggiornamento).

Argomento del Corso

Argomento del corso è la **programmazione** in un dato **linguaggio** (il *linguaggio C*) delle **macchine calcolatrici** (in particolare, i calcolatori digitali, o *computer*) affinché risolvano **problemi** per via automatica eseguendo **programmi** che mettono in atto (*implementano*) specifici **algoritmi** di risoluzione.

Argomento del Corso

Argomento del corso è la **programmazione** in un dato **linguaggio** (il *linguaggio C*) delle **macchine calcolatrici** (in particolare, i calcolatori digitali, o *computer*) affinché risolvano **problemi** per via automatica eseguendo **programmi** che mettono in atto (*implementano*) specifici **algoritmi** di risoluzione.

Parole Chiave:

- ❑ Problema
- ❑ Algoritmo
- ❑ Linguaggio
- ❑ Macchina calcolatrice
- ❑ Programmazione

Cominciamo dalla nozione di **algoritmo**, chiedendo lumi al Santo Patrono dei matematici.



Euclide (ca. 325 a.C. – ca. 265 a.C.)

Prologo

L'algoritmo euclideo

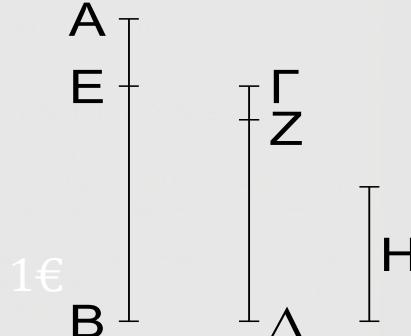
Nel VII Libro degli Elementi, Euclide dimostra costruttivamente che due numeri naturali qualunque hanno un massimo comun divisore (m.c.d):

1€

Nel VII Libro degli Elementi, Euclide dimostra costruttivamente che due numeri naturali qualunque hanno un massimo comun divisore (m.c.d):

β̄.

Δύο ἀριθμῶν δοθέντων μὴ πρώτων πρὸς ἀλλήλους τὸ μέγιστον αὐτῶν κοινὸν μέτρον εὑρεῖν.



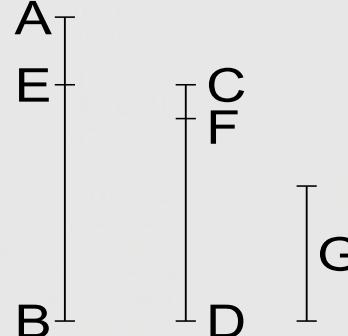
Ἐστωσαν οἱ δοθέντες δύο ἀριθμοὶ μὴ πρῶτοι πρὸς ἀλλήλους οἱ ΑΒ, ΓΔ. δεῖ δὴ τῶν ΑΒ, ΓΔ τὸ μέγιστον κοινὸν μέτρον εὑρεῖν.

Εἰ μὲν οὖν ὁ ΓΔ τὸν ΑΒ μετρεῖ, μετρεῖ δὲ καὶ ἑαυτόν, ὁ ΓΔ ἡρά τῶν ΓΔ, ΑΒ κοινὸν μέτρον ἔστιν. καὶ φανερόν, ὅτι καὶ μέγιστον οὐδεὶς γὰρ μείζων τοῦ ΓΔ τὸν ΓΔ μετρήσει.

Εἰ δὲ οὐ μετρεῖ ὁ ΓΔ τὸν ΑΒ, τῶν ΑΒ, ΓΔ ἀνθυφαιρουμένου ἀεὶ τοῦ ἐλάσσονος ἀπὸ τοῦ μείζονος λειφθήσεται τις ἀριθμός, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. μονὰς μὲν γὰρ οὐ λειφθήσεται· εἰ δὲ μῆ, ἔσονται οἱ ΑΒ, ΓΔ πρῶτοι πρὸς ἀλλήλους· ὅπερ οὐχ ὑπόκειται. λειφθήσεται τις ἡρά ἀριθμὸς, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. καὶ ὁ μὲν ΓΔ τὸν ΒΕ μετρῶν λειπέτω ἑαυτοῦ ἐλάσσονα τὸν ΕΑ, ὁ δὲ ΕΑ τὸν ΔΖ μετρῶν λειπέτω ἑαυτοῦ ἐλάσσονα τὸν ΖΓ, ὁ δὲ ΓΖ τὸν ΑΕ μετρείτω. ἐπεὶ οὖν ὁ ΓΖ τὸν ΑΕ μετρεῖ, ὁ δὲ ΑΕ τὸν ΔΖ μετρεῖ, καὶ ὁ ΓΖ ἡρά τὸν ΔΖ μετρήσει. μετρεῖ δὲ καὶ ἑαυτόν· καὶ ὅλον ἡρά τὸν ΓΔ μετρήσει. ὁ δὲ ΓΔ τὸν ΒΕ μετρεῖ καὶ ὁ ΓΖ ἡρά τὸν ΒΕ μετρεῖ μετρεῖ δὲ καὶ τὸν ΕΑ·

Proposition 2

To find the greatest common measure of two given numbers (which are) not prime to one another.



Let AB and CD be the two given numbers (which are) not prime to one another. So it is required to find the greatest common measure of AB and CD.

In fact, if CD measures AB, CD is thus a common measure of CD and AB, (since CD) also measures itself. And (it is) manifest that (it is) also the greatest (common measure). For nothing greater than CD can measure CD.

But if CD does not measure AB then some number will remain from AB and CD, the lesser being continually subtracted, in turn, from the greater, which will measure the (number) preceding it. For a unit will not be left. But if not, AB and CD will be prime to one another [Prop. 7.1]. The very opposite thing was assumed. Thus, some number will remain which will measure the (number) preceding it. And let CD measuring BE leave EA less than itself, and let EA measuring DF leave FC less than itself, and let CF measure AE. Therefore, since CF measures AE, and AE measures DF, CF will thus also measure DF. And it also measures itself. Thus, it will

La *procedura* descritta da Euclide si può riassumere in modo semi-formale – o, come anche si dice, in *pseudo-codice* – come segue. Si assuma che a e b siano due interi positivi.

1€

```
procedura mcd(a, b)
    finquando a ≠ b
        se a > b
            a := a - b
        altrimenti
            b := b - a
    restituisci a
```

La procedura di Euclide è un esempio di **algoritmo**: la prescrizione di una serie di operazioni “elementari” da compiere per risolvere un problema dato.

La Proposizione 2 del VII Libro degli Elementi dimostra che l’algoritmo euclideo è corretto e completo, come si dice in gergo: ossia, per ciascun valore possibile dei dati in ingresso (gli interi $a, b > 0$), termina restituendo il risultato corretto (il m.c.d. di a e b).

Domanda: Quanto rapidamente termina l'algoritmo euclideo in funzione dei dati in ingresso?

Per dare un senso un po' più preciso alla domanda, e rendere possibile una risposta, notiamo che le sottrazioni iterate dell'algoritmo sono equivalenti in modo ovvio a delle divisioni.

Denotiamo con

$$T(a,b)$$

il numero di divisioni richieste dall'algoritmo in funzione dei dati in ingresso a e b .

La *successione di Fibonacci* è: 1,1,2,3,5..., ossia $F_1=F_2=1$ e $F_n := F_{n-1}+F_{n-2}$ per $n>2$.

Domanda: Quanto rapidamente termina l'algoritmo euclideo in funzione dei dati in ingresso?

Per dare un senso un po' più preciso alla domanda, e rendere possibile una risposta, notiamo che le sottrazioni iterate dell'algoritmo sono equivalenti in modo ovvio a delle divisioni.

Denotiamo con

$$T(a,b)$$

il numero di divisioni richieste dall'algoritmo in funzione dei dati in ingresso a e b .

La *successione di Fibonacci* è: 1,1,2,3,5..., ossia $F_1=F_2=1$ e $F_n := F_{n-1}+F_{n-2}$ per $n>2$.

Proposizione (Gabriel Lamé, 1847). Sia $n > 0$ un intero, e siano $a > b > 0$ interi tali che l'algoritmo euclideo applicato ad a e b soddisfi $T(a,b)=n$. Se a è il minimo intero che soddisfa tali condizioni, si ha

$$a = F_{n+2}$$

$$b = F_{n+1}$$

Proposizione (Gabriel Lamé, 1847). Sia $n > 0$ un intero, e siano $a > b > 0$ interi tali che l'algoritmo euclideo applicato ad a e b soddisfi $T(a,b)=n$. Se a è il minimo intero che soddisfa tali condizioni, si ha

$$a = F_{n+2}$$

$$b = F_{n+1}$$

n	1	2	3	4	5	6	7	8	9	10	11	12	13
F_n	1	1	2	3	5	8	13	21	34	55	89	144	233

TABELLA 1. I primi termini della successione di Fibonacci

$a \setminus b$	1	2	3	4	5	6	7	8
1	1							
2	1	1						
3	1	2	1					
4	1	1	2	1				
5	1	2	3	2	1			
6	1	1	1	2	2	1		
7	1	2	2	3	3	2	1	
8	1	1	3	1	4	2	2	1

TABELLA 2. Il numero di divisioni eseguito dall'algoritmo euclideo

I termini della successione di Fibonacci ammettono una forma chiusa, nota col nome di *formula di Binet-de Moivre*:

$$F_{n-3} = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

dove

$$\phi := \frac{1+\sqrt{5}}{2} \approx 1.61803$$

è la *sezione aurea*.

I termini della successione di Fibonacci ammettono una forma chiusa, nota col nome di *formula di Binet-de Moivre*:

$$F_{n-3} = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

dove

$$\phi := \frac{1+\sqrt{5}}{2} \approx 1.61803$$

è la *sezione aurea*.

Usando la forma chiusa qui sopra, dalla proposizione dimostrata da Lamé si può ottenere:

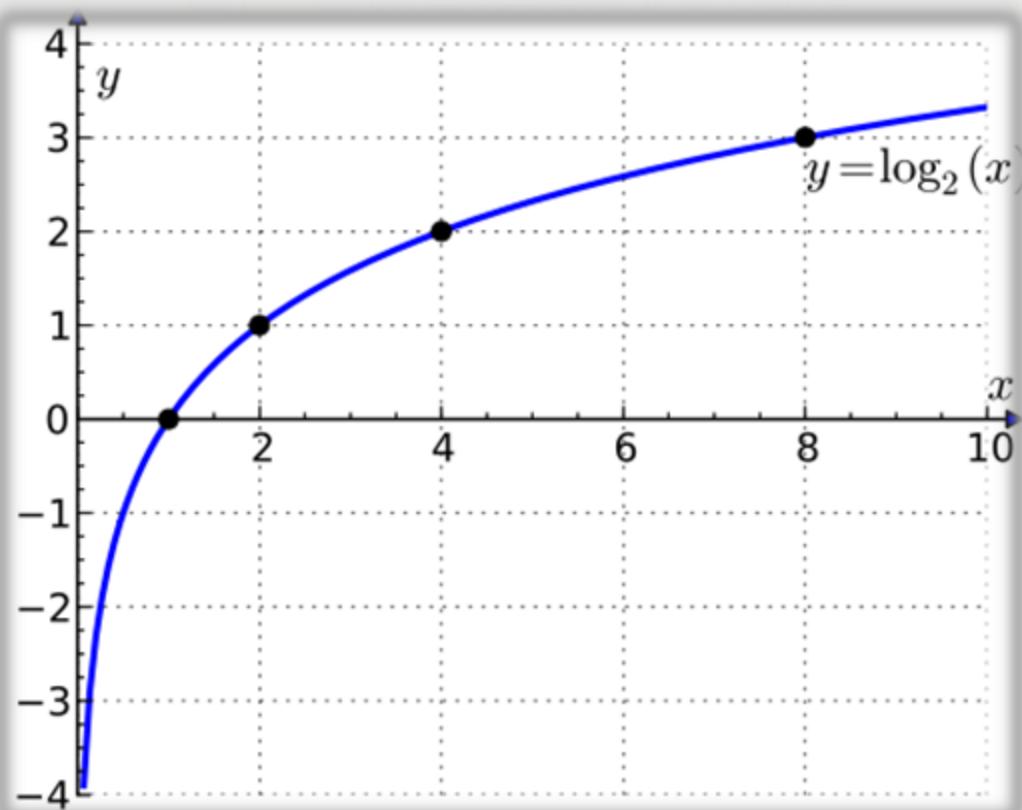
Corollario (Caso pessimo dell'algoritmo di euclide). Dati interi $a > b > 0$, si ha

$$T(a, b) \leq \lfloor \log_\phi(3 - \phi)a \rfloor$$

La funzione

$$\lfloor \log_{\phi}(3 - \phi)a \rfloor$$

cresce molto lentamente rispetto ad a . Per esempio, ecco la funzione logaritmo in base 2:



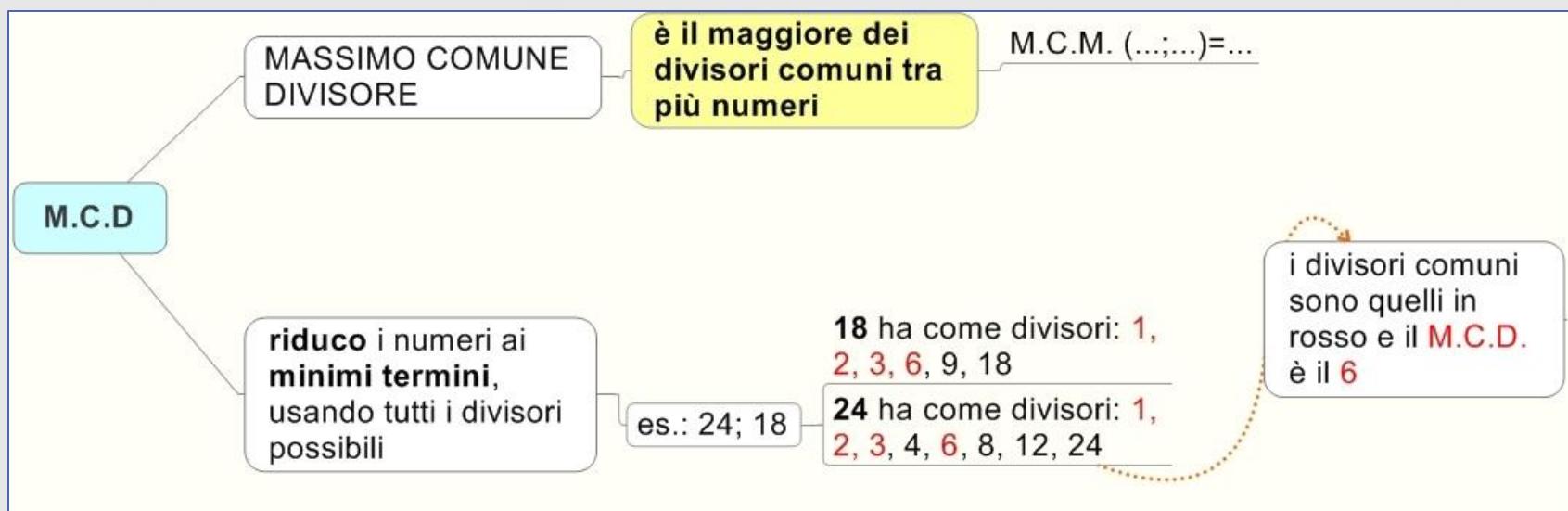
Informalmente: *L'algoritmo euclideo termina in “pochi passi”, rispetto alla magnitudine dei valori in ingresso.* In questo senso (impreciso), esso è **efficiente**.

Informalmente: *L'algoritmo euclideo termina in “pochi passi”, rispetto alla magnitudine dei valori in ingresso.* In questo senso (impreciso), esso è **efficiente**.

Confrontiamo queste conclusioni con un algoritmo che ancora oggi si insegna nelle scuole elementari per risolvere **il medesimo problema**. Da un sito web per le scuole elementari:

Informalmente: *L'algoritmo euclideo termina in “pochi passi”, rispetto alla magnitudine dei valori in ingresso.* In questo senso (impreciso), esso è **efficiente**.

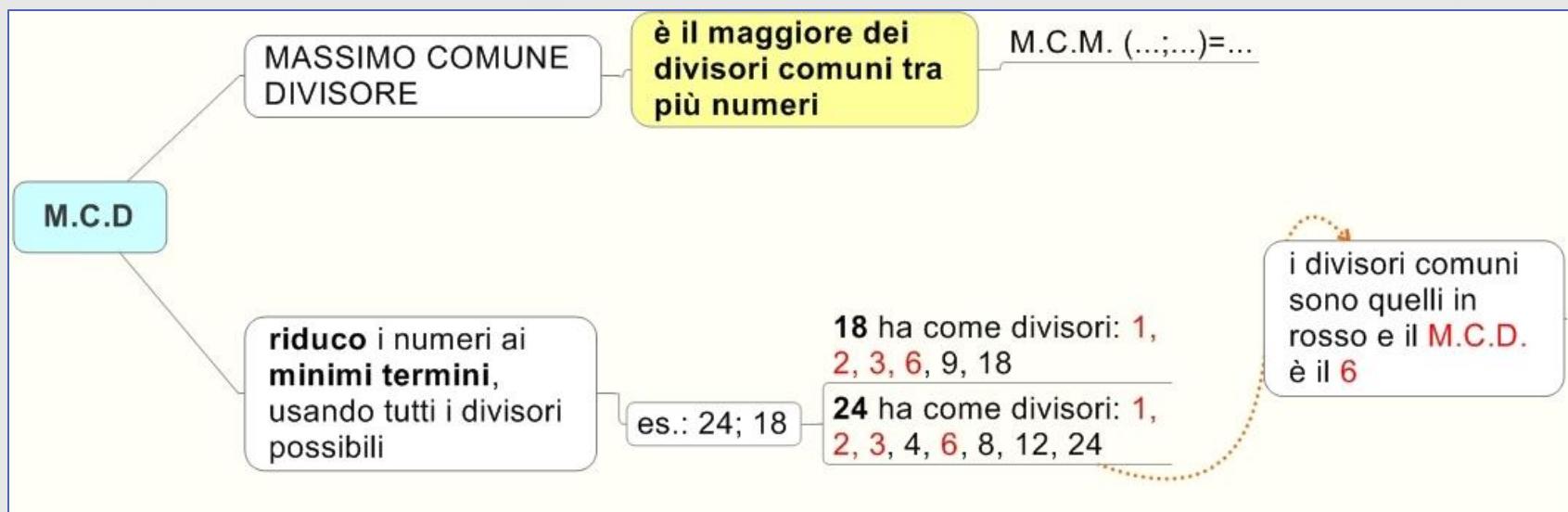
Confrontiamo queste conclusioni con un algoritmo che ancora oggi si insegna nelle scuole elementari per risolvere **il medesimo problema**. Da un sito web per le scuole elementari:



Tutto si riduce dunque a risolvere il problema ausiliario: **scomporre un intero positivo nei suoi fattori primi**.

Informalmente: *L'algoritmo euclideo termina in “pochi passi”, rispetto alla magnitudine dei valori in ingresso.* In questo senso (impreciso), esso è **efficiente**.

Confrontiamo queste conclusioni con un algoritmo che ancora oggi si insegna nelle scuole elementari per risolvere **il medesimo problema**. Da un sito web per le scuole elementari:



Tutto si riduce dunque a risolvere il problema ausiliario: **scomporre un intero positivo nei suoi fattori primi**.

Fatto Importante: *Non è noto alcun algoritmo efficiente per risolvere tale problema ausiliario.*

Un po' più precisamente: *Tutti gli algoritmi noti che risolvono il problema di scomporre un numero intero positivo x nei suoi fattori primi richiedono un numero di “passi” **super-polinomiale**: cioè, che cresce con x più di qualunque assegnata funzione polinomiale del numero di cifre (**bit**) necessarie per esprimere x nel sistema di numerazione binario.*

Un po' più precisamente: *Tutti gli algoritmi noti che risolvono il problema di scomporre un numero intero positivo x nei suoi fattori primi richiedono un numero di “passi” super-polinomiale*: cioè, che cresce con x più di qualunque assegnata funzione polinomiale del numero di cifre (**bit**) necessarie per esprimere x nel sistema di numerazione binario.

Il risvolto pratico di questi fatti è importante. La sicurezza di innumerevoli transazioni elettroniche con carta di credito, per esempio, dipende **crucialmente** dalla non-esistenza di un algoritmo efficiente per scomporre un numero intero nei suoi fattori primi.

In altri termini, la fattorizzazione in primi è un problema “difficile”. Non è per caso che, prima dell'avvento dei moderni computer, schiere di matematici si dedicavano a fattorizzare a mano numeri interi grandi – senza perdere il loro tempo, dal punto di vista concettuale, perché come abbiamo appena detto il problema è “difficile”. Per cominciare ad apprezzare *quanto* “difficile”, ascoltiamo un'altra storia.



David Hilbert (1862–1943)

“Problemi” e Problemi

Facili, difficili e irrisolubili

I numeri di Mersenne sono i numeri naturali della forma

$$M_p := 2^p - 1 ,$$

con p intero positivo *primo*. Nel XVII secolo, Padre Mersenne asserì che M_{67} era un numero primo. Lucas dimostrò invece nel 1876 che M_{67} deve avere fattori non banali, senza però riuscire ad esibirne alcuno.



Marin Mersenne (1588–1648)

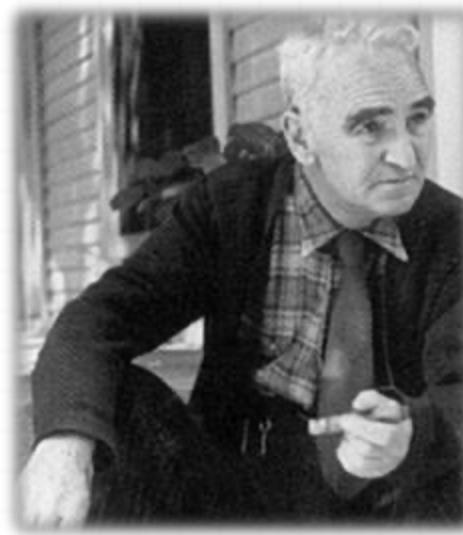


Édouard Lucas (1842–1891)

Nel 1903, il matematico statunitense Frank Cole riuscì per la prima volta a computare una fattorizzazione non banale di M_{67} . Egli comunicò il suo risultato in occasione di un incontro della Società Matematica Americana. Ecco come lo storico della matematica statunitense di origini scozzesi Eric T. Bell rievocava l'episodio molti anni dopo.



Frank N. Cole (1861–1926)



Eric T. Bell (1883–1960)

Vorrei qui tramandare un episodio storico, prima che tutti i matematici americani della prima metà del ventesimo secolo siano scomparsi. Quando, nel 1911, chiesi a Cole quanto avesse impiegato a fattorizzare M_{67} , mi rispose: "tre anni di domeniche". Sebbene interessante, non è questo l'episodio storico. Nella seduta della Società Matematica Americana che si tenne a New York nell'ottobre del 1903, Cole aveva una comunicazione in programma con il modesto titolo: "Sulla fattorizzazione dei numeri grandi". Quando il presidente annunciò il suo intervento, Cole – che non fu mai uomo di molte parole – si avvicinò alla lavagna e, in silenzio, calcolò il valore di 2 elevato alla sessantasettesima potenza. Con cura, sottrasse 1 al risultato. Senza dire una parola, si spostò quindi verso una porzione pulita di lavagna ed eseguì a mano la moltiplicazione

$$193.707.721 \times 761.838.257.287$$

I due conti concordavano. [...] [La] platea della Società Matematica Americana applaudì vigorosamente l'autore [...]. Cole riprese posto senza aver proferito parola. Nessuno gli pose domande.

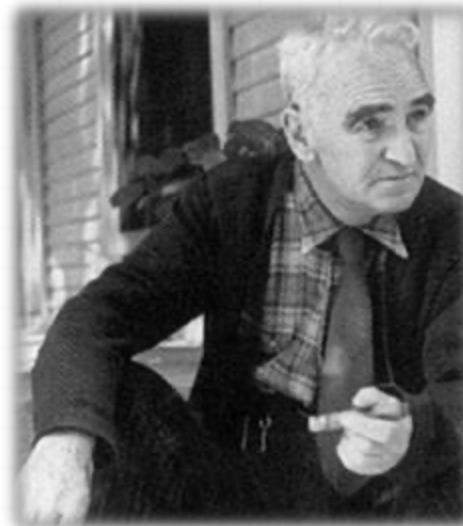
E. T. Bell. *Mathematics, Queen and Servant of Science*. 1951. p. 228. (Trad. mia.)

Oggi, un qualunque personal computer (adeguatamente programmato) è in grado di fattorizzare M_{67} in una manciata di secondi.

Sarebbe però un errore concettuale grave ritenere che il “problema” **generale** di cui Cole affrontò un caso particolare sia banale. Come abbiamo visto, non ci sono algoritmi “efficienti” per risolvere il “problema” generale della scomposizione di un numero intero nei suoi fattori primi. Si tratta di un “**problema difficile**”.



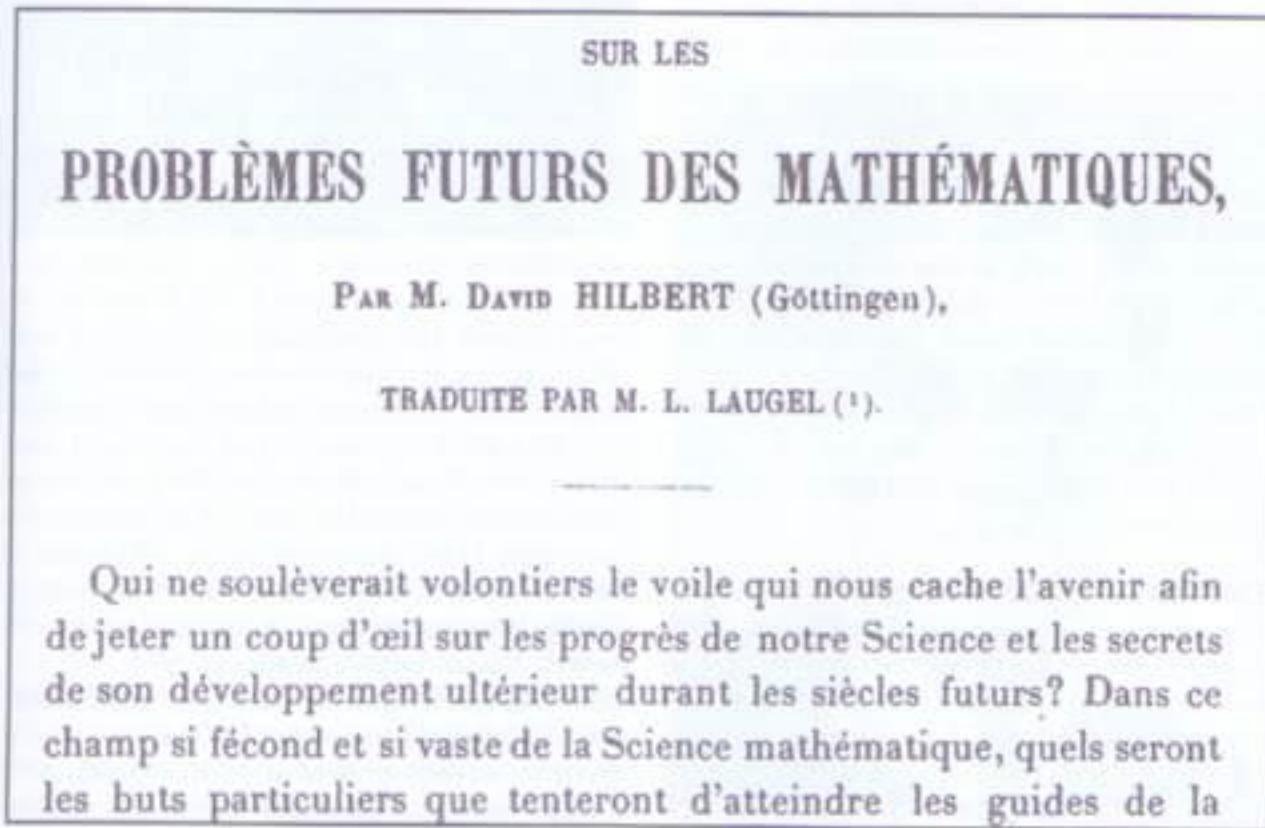
Frank N. Cole (1861–1926)



Eric T. Bell (1883–1960)

Finora abbiamo dunque visto che esistono “**problemi facili**” (*M.C.D.*), e “**problemi difficili**” (*fattorizzazione*). C’è dell’altro, però. Esistono anche dei problemi “**irrisolubili**”.

Finora abbiamo dunque visto che esistono “**problemi facili**” (*M.C.D.*), e “**problemi difficili**” (*fattorizzazione*). C’è dell’altro, però. Esistono anche dei problemi “**irrisolubili**”.



Hilbert's lecture from the Paris Proceedings

D. Hilbert, *I problemi futuri della matematica*, Atti del Congresso Internazionale dei Matematici, Parigi, 1900.

Finora abbiamo dunque visto che esistono “**problemi facili**” (*M.C.D.*), e “**problemi difficili**” (*fattorizzazione*). C’è dell’altro, però. Esistono anche dei problemi “**irrisolubili**”.

Un’**equazione diofantea** è un’equazione della forma:

$$p(x_1, \dots, x_n) = 0$$

dove p è un polinomio a coefficienti interi nelle variabili x_1, \dots, x_n .

Finora abbiamo dunque visto che esistono “**problemi facili**” (*M.C.D.*), e “**problemi difficili**” (*fattorizzazione*). C’è dell’altro, però. Esistono anche dei problemi “**irrisolubili**”.

Il Decimo Problema di Hilbert

Trovare una (singola) procedura che permetta, per ogni data equazione diofantea, di determinare tramite un numero finito di operazioni se l’equazione abbia o non abbia soluzioni intere.

Un’**equazione diofantea** è un’equazione della forma:

$$p(x_1, \dots, x_n) = 0$$

dove p è un polinomio a coefficienti interi nelle variabili x_1, \dots, x_n .

Finora abbiamo dunque visto che esistono “**problemi facili**” (*M.C.D.*), e “**problemi difficili**” (*fattorizzazione*). C’è dell’altro, però. Esistono anche dei problemi “**irrisolubili**”.

Il Decimo Problema di Hilbert

Trovare una (singola) procedura che permetta, per ogni data equazione diofantea, di determinare tramite un numero finito di operazioni se l’equazione abbia o non abbia soluzioni intere.

Per esempio, il metodo di risoluzione delle equazioni di secondo grado (a coefficienti in \mathbf{Z}) che abbiamo imparato a scuola costituisce una tale procedura nel caso particolare delle equazioni diofantee in una variabile e di grado al più 2: infatti, dalla espressione generale per le due soluzioni in \mathbf{C} , si determina agevolmente se ciascuna soluzione è in \mathbf{Z} .

Ma quanto abbiamo imparato a scuola sulle equazioni di secondo grado non si estende al caso generale.

“Teorema” (1970). Non esiste alcun algoritmo che risolva il Decimo Problema di Hilbert. Non esiste cioè alcun algoritmo che, avendo come dato in ingresso una qualsivoglia equazione diofantea, termini sempre, con risultato **SI** se l’equazione in ingresso ammette soluzioni intere, e con risultato **NO** se essa non ne ammette.

Ma quanto abbiamo imparato a scuola sulle equazioni di secondo grado non si estende al caso generale.

“**Teorema**” (1970). Non esiste alcun algoritmo che risolva il Decimo Problema di Hilbert. Non esiste cioè alcun algoritmo che, avendo come dato in ingresso una qualsivoglia equazione diofantea, termini sempre, con risultato **SI** se l’equazione in ingresso ammette soluzioni intere, e con risultato **NO** se essa non ne ammette.

Il “Teorema” è dovuto allo sforzo congiunto di molti matematici. È il *Teorema Matiyasevich/MRDP* di Matiyasevich, basato su risultati precedenti di Matiyasevich stesso, Robinson, Davis, e Putnam.

Ma quanto abbiamo imparato a scuola sulle equazioni di secondo grado non si estende al caso generale.

“Teorema” (1970). Non esiste alcun algoritmo che risolva il Decimo Problema di Hilbert. Non esiste cioè alcun algoritmo che, avendo come dato in ingresso una qualsivoglia equazione diofantea, termini sempre, con risultato **SI** se l'equazione in ingresso ammette soluzioni intere, e con risultato **NO** se essa non ne ammette.

Il “Teorema” è dovuto allo sforzo congiunto di molti matematici. È il *Teorema Matiyasevich/MRDP* di Matiyasevich, basato su risultati precedenti di Matiyasevich stesso, Robinson, Davis, e Putnam.

In matematica non esistono però “teoremi”, ma solo teoremi. Per rimuovere le virgolette dall'enunciato qui sopra, è necessario definire formalmente la nozione di **algoritmo**. Il fatto che ciò sia possibile, in modo utile e sensato, è una delle conquiste della scienza del XX secolo.



Macchine di Carta

La nozione di Passo di Calcolo

La difficoltà nella definizione formale della nozione di **algoritmo** risiede nella necessità di isolare con chiarezza la nozione ausiliaria, ma centrale in tutta la teoria della computazione, di **passo di calcolo elementare**.

Nell'algoritmo euclideo, cosa è da prendersi come passo elementare? Le singole sottrazioni? O le divisioni, come abbiamo fatto noi? O altro ancora?

E in un altro algoritmo per risolvere un problema diverso?

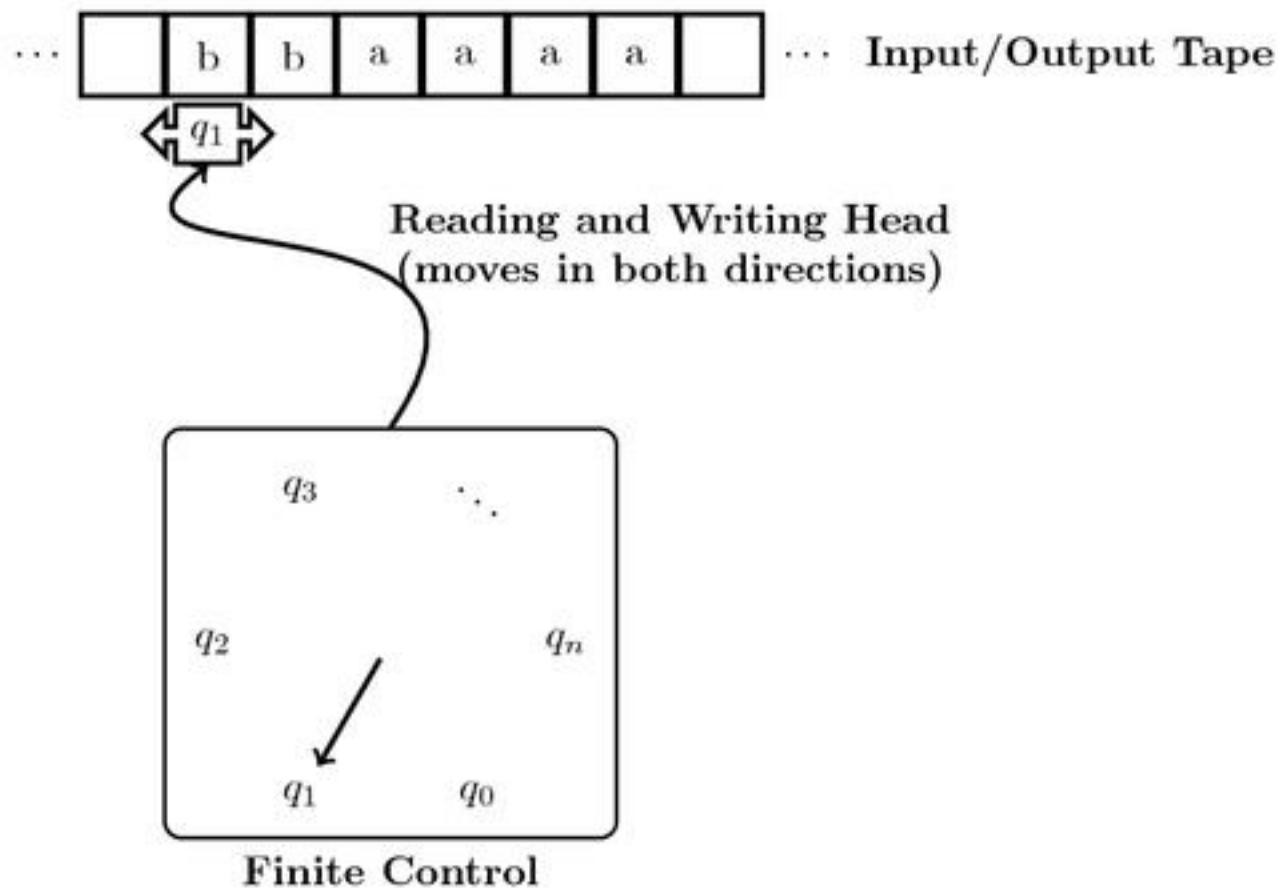
Se ciascun algoritmo si portasse dietro la sua propria nozione di passo elementare di calcolo, nulla rimarrebbe dell'idea intuitiva di algoritmo.

La definizione di algoritmo comporta la definizione di passo di calcolo.

Nel 1937, il matematico inglese Alan M. Turing, allora venticinquenne, pubblica sui *Proceedings of the London Mathematical Society* l'articolo: *On computable numbers, with an application to the Entscheidungsproblem*.

Turing identifica qui la nozione di passo di calcolo nel modo più semplice e diretto possibile, relativamente a un *modello astratto di macchina calcolatrice* oggi nota come **macchina di Turing**.

[Una macchina di Turing è costituita da] una capacità di memorizzazione [finita ma] illimitata, sotto forma di un nastro indefinitamente lungo suddiviso in caselle, su ciascuna delle quali si può scrivere un simbolo [estratto da un fissato alfabeto finito, per esempio l'alfabeto binario 0 e 1.] In ogni dato istante vi è un solo simbolo nella macchina; è detto “il simbolo letto”. La macchina può modificare il simbolo letto, e il suo comportamento è in parte determinato da quel simbolo, ma i rimanenti simboli sul nastro non influiscono sul comportamento della macchina. Il nastro, però, può essere fatto scorrere avanti e indietro attraverso [la testina di lettura e scrittura de]la macchina, e ciò costituisce una delle **operazioni elementari** della macchina. (Turing 1948; trad. mia; enfasi mia.)



Visualizzazione intuitiva di una macchina di Turing

Definizione formale (da Hopcroft e Ullman, 1979)

Una *macchina di Turing* è una settupla

$$(Q, \Gamma, b, \Sigma, \delta, q_0, F),$$

dove:

- Q è un insieme finito e non vuoto di *stati*.
- Γ è un insieme finito e non vuoto, detto *alfabeto (del nastro)*.
- $b \in \Gamma$ è il simbolo *spazio*.
- $\Sigma \subseteq \Gamma \setminus \{b\}$ è l'*insieme dei simboli in ingresso*.
- $q_0 \in Q$ è lo *stato iniziale*.
- $F \subseteq Q$ è l'insieme degli *stati finali*.
- $\delta: Q \setminus F \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ è una funzione parziale detta *funzione di transizione*, dove L sta per *spostamento di una casella del nastro a sinistra* ed R per *spostamento di una casella della nastro a destra*.

Cosa computa la macchina di Turing descritta dalla tabella?
(Configurazione iniziale del nastro: tutte le caselle contengono b.
Insieme degli stati finali: vuoto.)

Stato	Simbolo sul nastro	Operazione	Stato successivo
q_0	b	Scrivi 0, R	q_1
q_1	b	R	q_2
q_2	b	Scrivi 1, R	q_3
q_3	b	R	q_0

Cosa computa la macchina di Turing descritta dalla tabella?
 (Configurazione iniziale del nastro: tutte le caselle contengono b.
 Insieme degli stati finali: vuoto.)

Stato	Simbolo sul nastro	Operazione	Stato successivo
q_0	b	Scrivi 0, R	q_1
q_1	b	R	q_2
q_2	b	Scrivi 1, R	q_3
q_3	b	R	q_0

La macchina non termina mai, e continua a scrivere sul nastro, da sinistra a destra, la successione di numeri (separati da spazi):

0□1□0□1□0□1 ...

Un **algoritmo** è adesso definibile come *qualunque procedura che possa essere modellata da una macchina di Turing*.

Un **passo elementare di calcolo** dell'algoritmo è adesso definibile come *un'operazione elementare della macchina* (cfr. la citazione di Turing *supra*), e diviene una nozione matematica del tutto rigorosa: ad esempio, possiamo convenire che un **passo elementare** di una macchina di Turing è una singola applicazione della funzione δ nella definizione di Hopcroft e Ullman.

Delicate affermazioni di non esistenza, della forma “Non esiste un algoritmo (=macchina di Turing) tale che ...”, divengono così **suscettibili di dimostrazione matematica**. E tutto grazie, appunto, alla *definizione* di Turing: a riprova del fatto che, in matematica, le definizioni sono spesso più importanti dei teoremi.

Un **algoritmo** è adesso definibile come *qualunque procedura che possa essere modellata da una macchina di Turing*.

Un **passo elementare di calcolo** dell'algoritmo è adesso definibile come *un'operazione elementare della macchina* (cfr. la citazione di Turing *supra*), e diviene una nozione matematica del tutto rigorosa: ad esempio, possiamo convenire che un **passo elementare** di una macchina di Turing è una singola applicazione della funzione δ nella definizione di Hopcroft e Ullman.

Delicate affermazioni di non esistenza, della forma “Non esiste un algoritmo (=macchina di Turing) tale che ...”, divengono così **suscettibili di dimostrazione matematica**. E tutto grazie, appunto, alla *definizione* di Turing: a riprova del fatto che, in matematica, le definizioni sono spesso più importanti dei teoremi.

La soluzione negativa del Decimo Problema di Hilbert ammonta alla dimostrazione dell'impossibilità dell'esistenza di una macchina di Turing con determinate proprietà:

Teorema (1970). Non esiste alcun algoritmo che risolva il Decimo Problema di Hilbert. Non esiste cioè alcuna macchina di Turing che, avendo come dato in ingresso una qualsivoglia equazione diofantea, termini sempre, con risultato **SI** se l'equazione in ingresso ammette soluzioni intere, e con risultato **NO** se essa non ne ammette.

Si dice nel gergo della teoria della computazione che il problema di stabilire (algoritmicamente) se le equazioni diofantee ammettano soluzioni intere è **indecidibile**.

Si potrebbe compilare una *lunga* lista di problemi di interesse matematico che sono indecidibili nel senso appena illustrato. Noi però dobbiamo occuparci in questo corso di ciò che *si può* programmare, e non di ciò che *non si può* programmare.

Argomento del Corso

Argomento del corso è la **programmazione** in un dato **linguaggio** (il *linguaggio C*) delle **macchine calcolatrici** (in particolare, i calcolatori digitali, o *computer*) affinché risolvano **problemi** per via automatica eseguendo **programmi** che mettono in atto (*implementano*) specifici **algoritmi** di risoluzione.

Parole Chiave:

- ❑ Problema
- ❑ Algoritmo
- ❑ Linguaggio
- ❑ Macchina calcolatrice
- ❑ Programmazione

Dopo le macchine di Turing

- La storia che porta dalle macchine di Turing ai moderni calcolatori digitali è lunga e articolata. Potremo solo fare dei cenni a qualche pietra miliare.
- Il primo computer digitale “programmabile” è stato **Colossus**, entrato in funzione in Inghilterra nel 1944. Fu progettato e costruito dall’ingegnere Tommy Flowers in undici mesi, dietro richiesta e sotto la guida del grande topologo inglese Max Newmann, che era stato uno dei maestri di Turing a Cambridge.
- Nel 1942 Newmann aveva accettato di lavorare per il governo britannico a Bletchley Park, il centro militare per la decifrazione dei messaggi cifrati tedeschi.
- Gli inglesi avevano già decodificato i messaggi di **Enigma**, una macchina tedesca per cifrare messaggi. Colossus fu ideato e costruito per attaccare i messaggi della più recente macchina crittografica tedesca **Lorenz SZ40/42**.

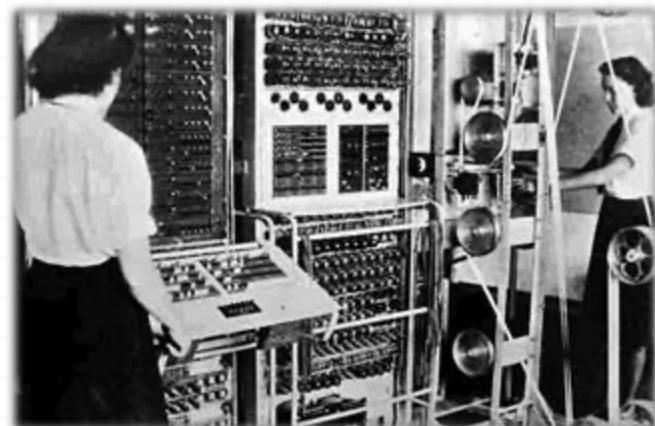
Dopo le macchine di Turing

- Colossus fu costruito usando un gran numero di valvole, un'idea che Flowers aveva già sperimentato anni prima nel suo lavoro civile per la compagnia telefonica britannica. I dati erano immessi tramite un nastro di carta. Era "programmabile" per poter eseguire una gamma di operazioni logiche sui dati in ingresso.
- Anche Turing lavorò a Bletchley park durante la guerra. Oggi sappiamo che il suo contributo alla decifrazione dei codici tedeschi fu di grande importanza.



Tomas H. Flowers, 1905–1998

Maxwell H. A. Newman, 1897–1984



Il Colossus a Bletchley Park, 1944

Dopo le macchine di Turing

- Nel frattempo, gli alleati americani non erano rimasti indietro. Il primo computer elettronico “programmabile” negli Stati Uniti fu **ENIAC** (*Electronic Numerical Integrator and Computer*).
- ENIAC fu progettato e costruito all’Università della Pennsylvania da un gruppo guidato da J. Mauchly e J. Presper Eckert. Il progetto cominciò nel 1943; ENIAC entrò in funzione in via definitiva nel 1945.



J. Presper Eckert, 1919–1995



John W. Mauchly, 1907–1980



John von Neumann, 1903–1957
(con, da sinistra, Ulam e Feynman)

L'architettura di Von Neumann

- Colossus ed ENIAC erano concepiti in modo simile, sebbene ENIAC fosse di parecchio più grosso. Nonostante li si possa considerare computer “programmabili”, in realtà sarebbe forse più corretto definirli “configurabili”. Un dato algoritmo era codificato all’interno di ENIAC da uno specifico stato dei suoi cavi e dei suoi interruttori. Implementare con ENIAC un algoritmo anche realtivamente semplice poteva richiedere settimane di lavoro.
- Un grande passo avanti avvenne quindi con la successiva generazione di calcolatori. Tralasciando i dettagli, l’idea di fondo è che *sia i dati sia le istruzioni del programma che deve manipolarli risiedano nella memoria del calcolatore* (in gergo, “**stored-program computer**”). In particolare, progettare un calcolatore secondo questo modello vuol dire anche fissare il suo **insieme di istruzioni**: concettualmente, i passi elementari di calcolo che esso può eseguire.
- Il matematico John von Neumann, ungherese espatriato negli Stati Uniti, lavorava alla costruzione della bomba all’idrogeno durante la guerra, nei laboratori di Los Alamos, in Nuovo Messico.

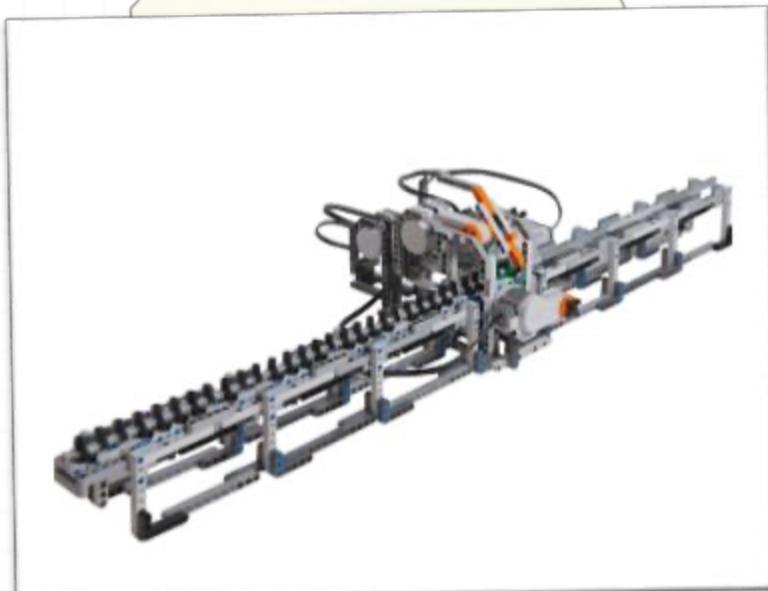
L'architettura di Von Neumann

- Von Neumann era in contatto con il gruppo che stava costruendo ENIAC all'Università della Pennsylvania. Conosceva inoltre bene il fondamentale lavoro di Turing del 1936.
- In effetti, nel suo lavoro Turing aveva concettualizzato in modo completo e dettagliato l'idea di un calcolatore programmabile a scopo generale, dimostrando l'esistenza di ciò che oggi chiamiamo una **Macchina di Turing Universale**, *in grado cioè di simulare il comportamento di qualunque altra macchina di Turing*.
- Turing usò quest'idea per esibire un semplice problema indecidibile, il **problema dell'arresto**, che diviene così il nostro secondo e ultimo esempio di problema indecidibile, dopo il Decimo di Hilbert.

Teorema (Turing 1936). Non esiste alcuna macchina di Turing che, ricevendo in ingresso (una codifica di) una qualsivoglia macchina di Turing M e una qualsivoglia quantità finita di dati D , decida se M si arresta ricevendo in ingresso D .

L'architettura di Von Neumann

- Mauchly ed Eckert, nell'agosto del 1944, cominciarono a lavorare al progetto di un nuovo calcolatore che superasse alcune limitazioni di ENIAC. Esso si sarebbe chiamato **EDVAC** (*Electronic Discrete Variable Automatic Computer*). EDVAC fu concepito secondo il modello genuinamente programmabile – e non meramente riconfigurabile – dello **stored-program computer**, cui si è accennato sopra.
- Quando von Neumann si unì al gruppo in qualità di consulente, si offrì di stendere un documento che riassumesse i principi di progettazione della nuova macchina.
- Lo fece in un lungo manoscritto, composto durante un viaggio di ritorno dalla Pennsylvania a Los Alamos. Il manoscritto, incompleto ma poi dattiloscritto e ampiamente diffuso, fu denominato ***First Draft of a Report on the EDVAC***.
- Fra molte altre cose, von Neumann spiegava qui l'architettura degli stored-program computer, che in essenza è quella usata ancora oggi. A causa di questo documento, ci si riferisce spesso all'architettura dei calcolatori moderni come all'**architettura di von Neumann**.



Macchina di Turing in Lego, 2012

Cenni all'architettura dei calcolatori

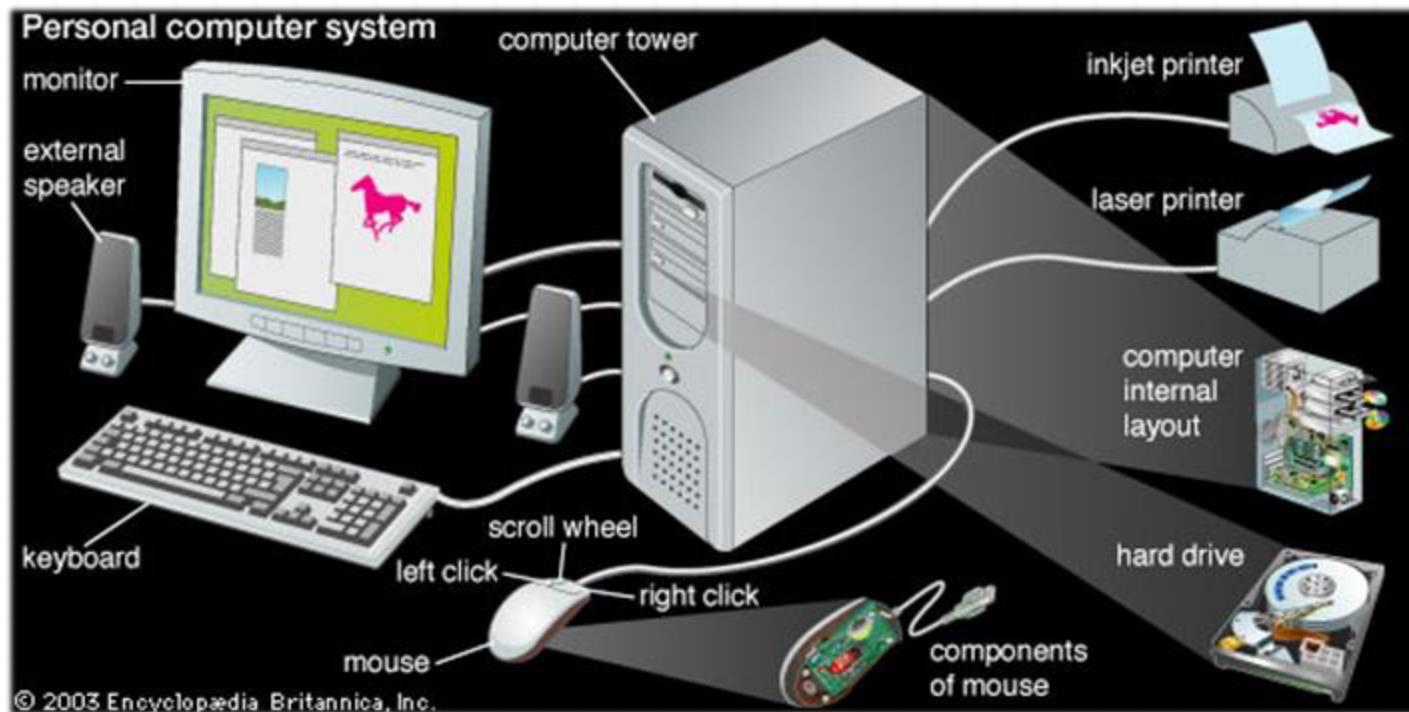
Il sostrato fisico della computazione nell'era digitale

Gli elementi fondamentali dell'architettura di un calcolatore sono i seguenti.

1. La **CPU** (Central Processing Unit), detta anche **(micro)processore**.
2. Il **bus** di sistema.
3. La **RAM** (Random Access Memory), detta anche **memoria centrale**.
4. La **memoria di massa**.
5. I dispositivi periferici per l'I/O, detti anche **periferiche** (monitor, tastiere...)

Gli elementi fondamentali dell'architettura di un calcolatore sono i seguenti.

1. La **CPU** (Central Processing Unit), detta anche **(micro)processore**.
2. Il **bus** di sistema.
3. La **RAM** (Random Access Memory), detta anche **memoria centrale**.
4. La **memoria di massa**.
5. I dispositivi periferici per l'I/O, detti anche **periferiche** (monitor, tastiere...)



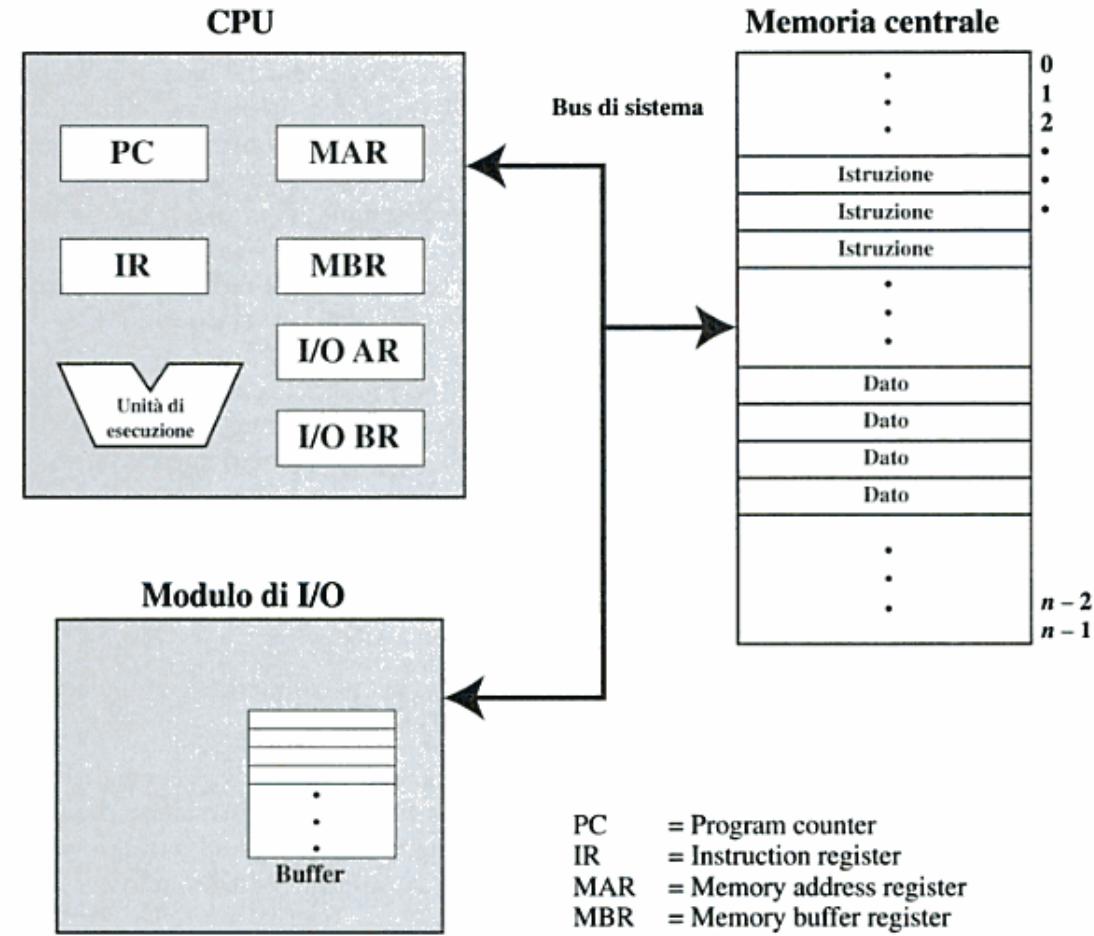


Figura 3.2 Componenti del calcolatore: visione ad alto livello.

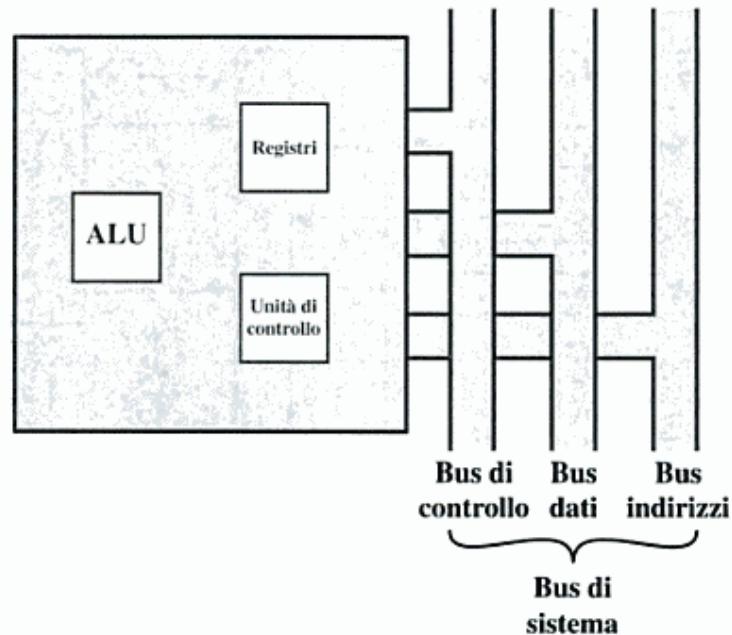


Figura 12.1 CPU e bus di sistema.

Da W. Stallings, *Architettura e organizzazione dei calcolatori*, Pearson Education Italia, 6a ed., 2004

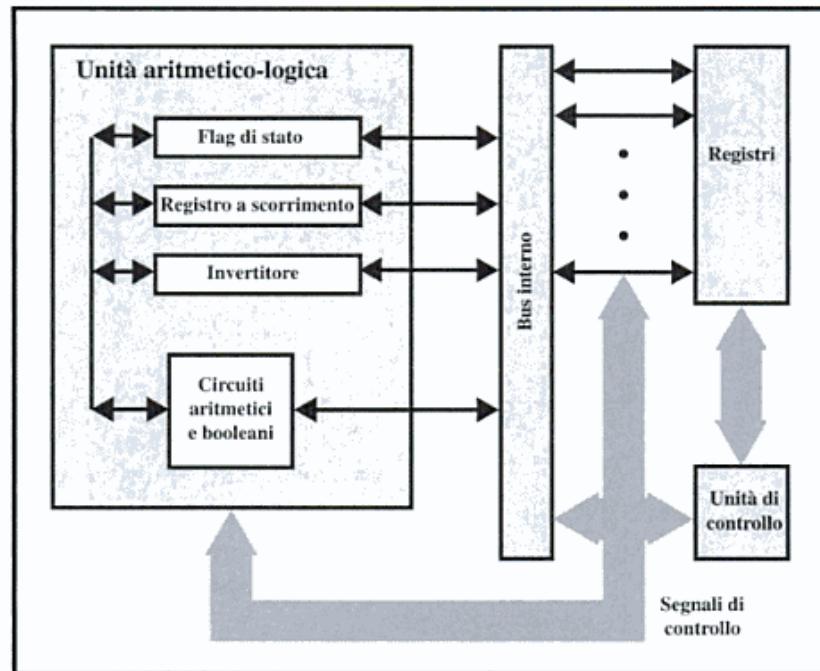


Figura 12.2 Struttura interna della CPU.

Da W. Stallings, *Architettura e organizzazione dei calcolatori*, Pearson Education Italia, 6a ed., 2004



La *Torre di Babele* di Brueghel il Vecchio, 1563

Linguaggi

La Babele digitale

La lingua madre del calcolatore è il **linguaggio macchina**: l'insieme di istruzioni che la CPU è in grado di *eseguire direttamente*.

Le istruzioni del linguaggio macchina sono codificate in notazione binaria, ed hanno un significato operazionale semplice come, per esempio: “**Sposta il contenuto del registro m nel registro n .**”

Un **programma in linguaggio macchina** è un insieme (finito, naturalmente) di tali istruzioni.

La lingua madre del calcolatore è il **linguaggio macchina**: l'insieme di istruzioni che la CPU è in grado di *eseguire direttamente*.

Le istruzioni del linguaggio macchina sono codificate in notazione binaria, ed hanno un significato operazionale semplice come, per esempio: “**Sposta il contenuto del registro m nel registro n .**”

Un **programma in linguaggio macchina** è un insieme (finito, naturalmente) di tali istruzioni.

Esempio: Linguaggio macchina MIPS

```
0000000100101010010000000100000
```

La lingua madre del calcolatore è il **linguaggio macchina**: l'insieme di istruzioni che la CPU è in grado di *eseguire direttamente*.

Le istruzioni del linguaggio macchina sono codificate in notazione binaria, ed hanno un significato operazionale semplice come, per esempio: “**Sposta il contenuto del registro *m* nel registro *n*.**”

Un **programma in linguaggio macchina** è un insieme (finito, naturalmente) di tali istruzioni.

Esempio: Linguaggio macchina MIPS

0000000100101010010000000100000

Campi: 000000|01001|01010|01000|00000|100000

OpCode|Reg Sr1|Reg Sr2|Reg Dst|Shift|Funz

La lingua madre del calcolatore è il **linguaggio macchina**: l'insieme di istruzioni che la CPU è in grado di *eseguire direttamente*.

Le istruzioni del linguaggio macchina sono codificate in notazione binaria, ed hanno un significato operazionale semplice come, per esempio: “**Sposta il contenuto del registro *m* nel registro *n*.**”

Un **programma in linguaggio macchina** è un insieme (finito, naturalmente) di tali istruzioni.

Esempio: Linguaggio macchina MIPS

0000000100101010010000000100000

Campi: 000000|01001|01010|01000|00000|100000

OpCode|Reg Sr1|Reg Sr2|Reg Dst|Shift|Funz

Significato: Somma i contenuti dei registri 9 e 10, e metti il risultato nel registro 8

Poter programmare un calcolatore in linguaggio macchina, come s'è già detto, è un enorme passo avanti rispetto, diciamo, alla prospettiva di dover riconfigurare un ENIAC.

La programmazione in linguaggio macchina, però, rimane pur sempre drammaticamente noiosa e soggetta ad errori materiali.

Le cose si possono migliorare di molto con un vecchio cavallo di battaglia dei matematici: un **cambio di notazione**.

Esempio: Linguaggio macchina MIPS

0000000100101010010000000100000

Campi: 000000|01001|01010|01000|00000|100000
OpCode|Reg Sr1|Reg Sr2|Reg Dst|Shift|Funz

Significato: Somma i contenuti dei registri 9 e 10, e metti il risultato nel registro 8

Poter programmare un calcolatore in linguaggio macchina, come s'è già detto, è un enorme passo avanti rispetto, diciamo, alla prospettiva di dover riconfigurare un ENIAC.

La programmazione in linguaggio macchina, però, rimane pur sempre drammaticamente noiosa e soggetta ad errori materiali.

Le cose si possono migliorare di molto con un vecchio cavallo di battaglia dei matematici: un **cambio di notazione**.

Esempio: Linguaggio macchina MIPS

0000000100101010010000000100000

Campi: 000000|01001|01010|01000|00000|100000
OpCode|Reg Sr1|Reg Sr2|Reg Dst|Shift|Funz

Significato: Somma i contenuti dei registri 9 e 10, e metti il risultato nel registro 8

add \$8, \$9, \$10

Poter programmare un calcolatore in linguaggio macchina, come s'è già detto, è un enorme passo avanti rispetto, diciamo, alla prospettiva di dover riconfigurare un ENIAC.

La programmazione in linguaggio macchina, però, rimane pur sempre drammaticamente noiosa e soggetta ad errori materiali.

Le cose si possono migliorare di molto con un vecchio cavallo di battaglia dei matematici: un **cambio di notazione**.

Esempio: Linguaggio macchina MIPS

0000000100101010010000000100000

Campi: 000000|01001|01010|01000|00000|100000

OpCode|Reg Sr1|Reg Sr2|Reg Dst|Shift|Funz

Significato: Somma i contenuti dei registri 9 e 10, e metti il risultato nel registro 8

add \$8, \$9, \$10

add \$t0, \$t1, \$t2

Poter programmare un calcolatore in linguaggio macchina, come s'è già detto, è un enorme passo avanti rispetto, diciamo, alla prospettiva di dover riconfigurare un ENIAC.

La programmazione in linguaggio macchina, però, rimane pur sempre drammaticamente noiosa e soggetta ad errori materiali.

Le cose si possono migliorare di molto con un vecchio cavallo di battaglia dei matematici: un **cambio di notazione**.

Esempio: Linguaggio macchina MIPS

0000000100101010010000000100000

Campi: 000000|01001|01010|01000|00000|100000
OpCode|Reg Sr1|Reg Sr2|Reg Dst|Shift|Funz

Significato: Somma i contenuti dei registri 8 e 9, e metti il risultato nel registro 10

Codice Assembly:

```
add $8, $9, $10  
add $t0, $t1, $t2
```

Il **codice** o **linguaggio assembly** è una traduzione mnemonica del linguaggio macchina. Sebbene nella pratica le cose siano un po' più intricate, concettualmente è corretto assumere l'esistenza di una *biiezione* che associa ciascuna istruzione assembly alla sua corrispondente istruzione macchina, e viceversa.

L'esperienza dimostra che questo semplice cambio di notazione facilita enormemente l'attività del programmatore.

Naturalmente, rimane un problema: chi si occuperà di tradurre per la CPU il codice assembly in codice macchina? Ossia, chi **computerà** la biiezione di cui sopra?

Il **codice** o **linguaggio assembly** è una traduzione mnemonica del linguaggio macchina. Sebbene nella pratica le cose siano un po' più intricate, concettualmente è corretto assumere l'esistenza di una *biiezione* che associa ciascuna istruzione assembly alla sua corrispondente istruzione macchina, e viceversa.

L'esperienza dimostra che questo semplice cambio di notazione facilita enormemente l'attività del programmatore.

Naturalmente, rimane un problema: chi si occuperà di tradurre per la CPU il codice assembly in codice macchina? Ossia, chi **computerà** la biiezione di cui sopra?

Risposta. La traduzione è eseguita per via automatica da un opportuno programma, scritto in codice macchina una volta per tutte.

Questo programma-traduttore si chiama **assemblatore** (o **assembler**).

Si noti che l'assemblatore si comporta come un **traduttore**, non come un **interprete (simultaneo)**: esso riceve in ingresso l'*intero* testo da tradurre (=il programma scritto in assembly) e produce in uscita il testo tradotto (=il programma in linguaggio macchina eseguibile dalla CPU).

Il processo di traduzione eseguito dall'assemblatore è detto **assemblaggio**, o, con un termine generico che si applica anche ad altri contesti, **compilazione**.

Grazie all'architettura di von Neumann, il programma assembly da tradurre **può risiedere nella memoria centrale**, al pari di qualunque altro dato e programma, **incluso l'assemblatore**.

Per scrivere materialmente il programma assembly si usano programmi ausiliari, gli **editor** (=semplici programmi per l'elaborazione dei testi.)

Una volta capito il trucco, però, ci vuol poco ad annoiarsi anche della programmazione in assembly: e si generalizza.

Possiamo concepire un linguaggio di programmazione le cui istruzioni abbiano un significato operazionale più complesso di quelle tipiche dell'assembly e del codice macchina.

Queste istruzioni più astratte – o di più alto livello, come si dice in gergo informatico – potranno poi essere tradotte in opportune sequenze di istruzioni assembly, e da lì in codice macchina.

Nascono così i **linguaggi di programmazione moderni**.

Come per le lingue naturali, vi sono molti più linguaggi di programmazione in uso di quanti non sia possibile anche solo menzionare qui. Citiamo solo tre linguaggi storicamente importanti, e tuttora in uso.

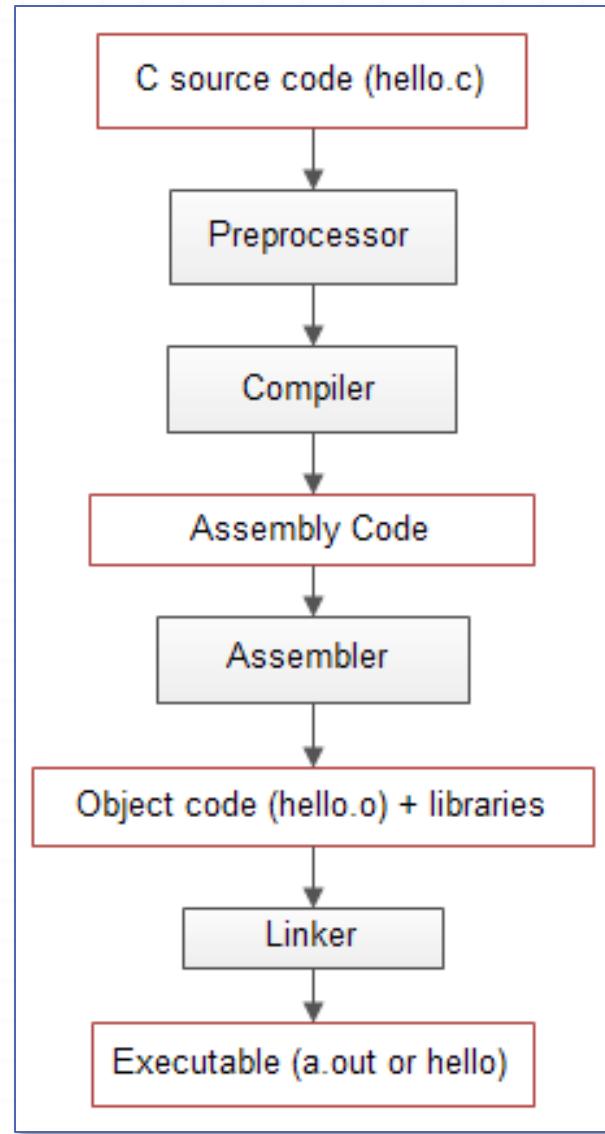
- 1957: **FORTRAN** (Formula Translating System). Pensato per applicazioni al calcolo scientifico. Inventore principale: John Backus. Finanziatore: IBM.
- 1959: **COBOL** (Common Business-Oriented Language). Pensato per applicazioni gestionali e finanziarie. Inventrice principale: Grace Hopper. Finanziatori: Un consorzio di industrie e università statunitensi, cofinanziate dal governo.
- 1969–1972: **C**. Pensato come strumento per la programmazione di sistema all'interno del sistema operativo UNIX. Inventori principali: Dennis Ritchie e Ken Thompson.

- 1957: FORTRAN (Formula Translating System). Pensato per applicazioni al calcolo scientifico. Inventore principale: John Backus. Finanziatore: IBM.
- 1959: COBOL (Common Business-Oriented Language). Pensato per applicazioni gestionali e finanziarie. Inventrice principale: Grace Hopper. Finanziatori: Un consorzio di industrie e università statunitensi, cofinanziate dal governo.
- 1969–1972: C. Pensato come strumento per la programmazione di sistema all'interno del sistema operativo UNIX. Inventori principali: Dennis Ritchie e Ken Thompson.



K. Thompson e D. Ritchie

- 1. Tassonomia.** I linguaggi si possono categorizzare in molti modi diversi: compilati vs. interpretati, alto livello vs. basso livello, imperativi vs. dichiarativi, procedurali vs. orientati agli oggetti, e così via *ad libitum*. Come descrizione di massima: **Il C è un linguaggio compilato, imperativo, procedurale, di basso livello.**
- 2. Standard.** Tutti i linguaggi di programmazione di uso corrente hanno una definizione formale precisa. Il linguaggio C non fa eccezione. Il libro di testo del corso si basa sullo *standard ANSI C* del 1989-1990, cui ci si riferisce come **C90**, o a volte come **C89**. Le versioni correnti del compilatore `gcc` rispettano tipicamente il successivo standard **C99**, del 1999. Ai nostri fini non è importante far riferimento allo standard più recente del 2011, il **C11**, che pochi compilatori implementano compiutamente. **Faremo di solito riferimento allo standard C99.**



Il ciclo di compilazione in C