# Informatica teorica

## Indice

# 1. Lezione 01

## 1.1. Introduzione

### 1.1.1. Definizione

L'**informatica teorica** è quella branca dell'informatica che si "contrappone" all'informatica applicata: in quest'ultima, l'informatica è usata solo come uno *strumento* per gestire l'oggetto del discorso, mentre nella prima l'informatica diventa l'*oggetto* del discorso, di cui ne vengono studiati i fondamenti.

### 1.1.2. Cosa e come

Analizziamo i due aspetti fondamentali che caratterizzano ogni materia:

1. il **cosa**: l'informatica è la scienza che studia l'informazione e la sua elaborazione automatica mediante un sistema di calcolo. Ogni volta che ho un *problema* cerco di risolverlo automaticamente scrivendo un programma. *Posso farlo per ogni problema? Esistono problemi che non sono risolubili?* Possiamo chiamare questo primo aspetto con il nome di **teoria della calcolabilità**, quella branca che studia cosa è calcolabile e cosa non lo è, a prescindere dal costo in termini di risorse che ne deriva. In questa parte utilizzeremo una caratterizzazione molto rigorosa e una definizione di problema il più generale possibile, così che l'analisi non dipenda da fattori tecnologici, storici...
2. il **come**: è relazionato alla **teoria della complessità**, quella branca che studia la quantità di risorse computazionali richieste nella soluzione automatica di un problema. Con *risorsa computazionale* si intende qualsiasi cosa venga consumato durante l'esecuzione di un programma, ad esempio:
   - elettricità;
   - numero di processori;
   - tempo;
   - spazio di memoria.

   In questa parte daremo una definizione rigorosa di tempo, spazio e di problema efficientemente risolubile in tempo e spazio, oltre che uno sguardo al famoso problema `P = NP`.

Possiamo dire che il *cosa* è uno studio **qualitativo**, mentre il *come* è uno studio **quantitativo**.

Grazie alla teoria della calcolabilità individueremo le funzioni calcolabili, di cui studieremo la complessità.

## 1.2. Richiami matematici

### 1.2.1. Funzioni

Una **funzione** da un insieme $A$ ad un insieme $B$ è una *legge*, spesso indicata con $f$, che spiega come associare agli elementi di $A$ un elemento di $B$.

Abbiamo due tipi di funzioni:

- **generale**: la funzione è definita in modo generale come $f : A \longrightarrow B$, in cui $A$ è detto **dominio** di $f$ e $B$ è detto **codominio** di $f$;
- **locale/puntuale**: la funzione riguarda i singoli valori $a$ e $b$:

$$f(a) = b \quad | \quad a \overset{f}{\longmapsto} b$$

  in cui $b$ è detta **immagine** di $a$ rispetto ad $f$ e $a$ è detta **controimmagine** di $b$ rispetto ad $f$.

Possiamo categorizzare le funzioni in base ad alcune proprietà:

- **iniettività**: una funzione $f : A \longrightarrow B$ si dice *iniettiva* se e solo se:

$$\forall a_1, a_2 \in A \quad a_1 \neq a_2 \Longrightarrow f(a_1) \neq f(a_2)$$

In poche parole, non ho *confluenze*, ovvero *elementi diversi finiscono in elementi diversi.*

- **suriettività**: una funzione $f : A \longrightarrow B$ si dice *suriettiva* se e solo se:

$$\forall b \in B \quad \exists a \in A \mid f(a) = b.$$

In poche parole, *ogni elemento del codominio ha almeno una controimmagine.*

Se definiamo l'**insieme immagine**:

$$\mathrm{Im}_f = \{b \in B \mid \exists a \in A \text{ tale che } f(a) = b\} = \{f(a) \mid a \in A\} \subseteq B$$

possiamo dare una definizione alternativa di funzione suriettiva, in particolare una funzione è *suriettiva* se e solo se $\mathrm{Im}_f = B$.

Infine, una funzione $f : A \longrightarrow B$ si dice **biiettiva** se e solo se è iniettiva e suriettiva, quindi vale: $\forall b \in B \quad \exists! a \in A \mid f(a) = b$.

Se $f : A \longrightarrow B$ è una funzione biiettiva, si definisce **inversa** di $f$ la funzione $f^{-1} : B \longrightarrow A$ tale che:

$$f(a) = b \iff f^{-1}(b) = a.$$

Per definire la funzione inversa $f^{-1}$, la funzione $f$ deve essere biiettiva: se così non fosse, la sua inversa avrebbe problemi di definizione.

Un'operazione definita su funzioni è la **composizione**: date $f : A \longrightarrow B$ e $g : B \longrightarrow C$, la funzione $f$ *composto* $g$ è la funzione $g \circ f : A \longrightarrow C$ definita come $(g \circ f)(a) = g(f(a))$.

La composizione *non è commutativa*, quindi $g \circ f \neq f \circ g$ in generale, ma è *associativa*, quindi $(f \circ g) \circ h = f \circ (g \circ h)$.

La composizione *f composto g* la possiamo leggere come *prima agisce f poi agisce g*.

Dato l'insieme $A$, la **funzione identità** su $A$ è la funzione $i_A : A \longrightarrow A$ tale che:

$$i_A(a) = a \quad \forall a \in A,$$

ovvero è una funzione che mappa ogni elemento su se stesso.

Grazie alla funzione identità, possiamo dare una definizione alternativa di funzione inversa: data la funzione $f : A \longrightarrow B$ biiettiva, la sua inversa è l'unica funzione $f^{-1} : B \longrightarrow A$ che soddisfa:

$$f \circ f^{-1} = f^{-1} \circ f = \mathrm{id}_A .$$

Possiamo vedere $f^{-1}$ come l'unica funzione che ci permette di *tornare indietro.*

# 2. Lezione 02

## 2.1. Richiami matematici

### 2.1.1. Funzioni totali e parziali

Prima di fare un'ulteriore classificazione per le funzioni, introduciamo la notazione $f(a)\downarrow$ per indicare che la funzione $f$ è definita per l'input $a$, mentre la notazione $f(a)\uparrow$ per indicare la situazione opposta.

Ora, data $f : A \longrightarrow B$ diciamo che $f$ è:
- **totale** se è definita *per ogni elemento* $a \in A$, ovvero $f(a)\downarrow \ \forall a \in A$;
- **parziale** se è definita *per qualche elemento* $a \in A$, ovvero $\exists a \in A \mid f(a)\uparrow$.

Chiamiamo **dominio** (o *campo*) **di esistenza** di $f$ l'insieme

$$\mathrm{Dom}_f = \{a \in A \mid f(a)\downarrow\} \subseteq A.$$

Notiamo che:
- $\mathrm{Dom}_f \subsetneq A \Longrightarrow f$ parziale;
- $\mathrm{Dom}_f = A \Longrightarrow f$ totale.

### 2.1.2. Totalizzare una funzione

È possibile *totalizzazione* una funzione parziale definendo una funzione a tratti $f : A \longrightarrow B \cup \{\perp\}$ tale che

$$\overline{f}(a) = \begin{cases} f(a) & a \in \mathrm{Dom}_f(a) \\ \perp & \text{altrimenti} \end{cases}.$$

Il nuovo simbolo introdotto è il *simbolo di indefinito*, e viene utilizzato per tutti i valori per cui la funzione di partenza $f$ non è appunto definita.

Da qui in avanti utilizzeremo $B_\perp$ come abbreviazione di $B \cup \{\perp\}$.

### 2.1.3. Prodotto cartesiano

Chiamiamo **prodotto cartesiano** l'insieme

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\},$$

che rappresenta l'insieme di tutte le *coppie ordinate* di valori in $A$ e in $B$.

In generale, il prodotto cartesiano **non è commutativo**, a meno che $A = B$.

Possiamo estendere il concetto di prodotto cartesiano a $n$-uple di valori, ovvero

$$A_1 \times ... \times A_n = \{(a_1, ..., a_n) \mid a_i \in A_i\}.$$

L'operazione "*opposta*" è effettuata dal **proiettore** $i$-esimo: esso è una funzione che estrae l'$i$-esimo elemento di un tupla, ovvero è una funzione $\pi_i : A_1 \times ... \times A_n \longrightarrow A_i$ tale che

$$\pi_i(a_1, ..., a_n) = a_i$$

Per comodità chiameremo $\underbrace{A \times ... \times A}_{n} = A^n$

### 2.1.4. Insieme di funzioni

L'insieme di tutte le funzioni da $A$ in $B$ si indica con

$$B^A = \{f : A \longrightarrow B\}.$$

Viene utilizzata questa notazione in quanto la cardinalità di $B^A$ é esattamente $|B|^{|A|}$, se $A$ e $B$ sono insiemi finiti.

In questo insieme sono presenti anche tutte le funzioni parziali da $A$ in $B$: mettiamo in evidenza questa proprietà, scrivendo

$$B_\perp^A = \{f : A \to B_\perp\}.$$

Sembrano due insiemi diversi ma sono la stessa cosa: nel secondo viene messo in evidenza il fatto che tutte le funzioni che sono presenti sono totali oppure parziali che sono state totalizzate.

#### 2.1.5. Funzione di valutazione

Dati $A, B$ e $B_\perp^A$ si definisce **funzione di valutazione** la funzione

$$\omega : B_\perp^A \times A \longrightarrow B$$

tale che

$$w(f, a) = f(a).$$

In poche parole, è una funzione che prende una funzione $f$ e la valuta su un elemento $a$ del dominio.

Abbiamo due possibili approcci:
- tengo fisso $a$ e provo tutte le funzioni $f$: sto eseguendo un *benchmark*, quest'ultimo rappresentato da $a$;
- tengo fissa $f$ e provo tutte le $a$ del dominio: sto ottenendo il grafico di $f$.

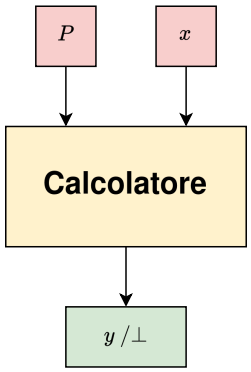
## 2.2. Teoria della calcolabilità

#### 2.2.1. Sistema di calcolo

Quello che faremo ora è *modellare teoricamente un sistema di calcolo.*

Un **sistema di calcolo** lo possiamo vedere come una *black-box* che prende un programma $P$, una serie di dati $x$ e calcola il risultato di $P$ sull'input $x$.

La macchina ci restituisce:
- $y$ se è riuscita a calcolare un risultato;
- $\perp$ se è entrata in un loop.




Un sistema di calcolo quindi è una funzione del tipo

$$\mathcal{C} : \text{PROG} \times \text{DATI} \longrightarrow \text{DATI}_\perp.$$

Come vediamo, è molto simile ad una funzione di valutazione: infatti, la parte dei dati $x$ la riusciamo a "mappare" sull'input $a$ del dominio, ma facciamo più fatica con l'insieme dei programmi, questo perché non abbiamo ancora definito cos'é un programma.

Un **programma** $P$ è una **sequenza di regole** che trasformano un dato di input in uno di output (o in un loop): in poche parole, un programma è una funzione del tipo

$$P : \mathrm{DATI} \longrightarrow \mathrm{DATI}_\perp,$$

ovvero una funzione che appartiene all'insieme $\mathrm{DATI}_\perp^{\mathrm{DATI}}$.

Con questa definizione riusciamo a "mappare" l'insieme PROG sull'insieme delle funzioni che ci serviva per definire la funzione di valutazione.

Formalmente, un sistema di calcolo è quindi la funzione

$$\mathcal{C} : \mathrm{DATI}_\perp^{\mathrm{DATI}} \times \mathrm{DATI} \longrightarrow \mathrm{DATI}.$$

Con $\mathcal{C}(P, x)$ indichiamo la funzione calcolata da $P$ su $x$, ovvero la sua **semantica**, il suo *significato*.

Il modello classico che viene considerato quando si parla di calcolatori è quello di **Von Neumann**.

### 2.2.2. Potenza computazionale

Prima di definire la potenza computazionale facciamo una breve premessa: indichiamo con

$$\mathcal{C}(P, \_) : \mathrm{DATI} \longrightarrow \mathrm{DATI}_\perp$$

la funzione che viene calcolata dal programma $P$, ovvero la semantica di $P$.

Fatta questa premessa, definiamo la **potenza computazionale** di un calcolatore come l'insieme di tutte le funzioni che quel sistema di calcolo può calcolare grazie ai programmi, ovvero

$$F(\mathcal{C}) = \{\mathcal{C}(P, \_) \mid P \in \mathrm{PROG}\} \subseteq \mathrm{DATI}_\perp^{\mathrm{DATI}}.$$

In poche parole, $F(\mathcal{C})$ rappresenta l'insieme di tutte le possibili semantiche di funzioni che sono calcolabili con il sistema $\mathcal{C}$.

Stabilire *cosa può fare l'informatica* equivale a studiare quest'ultima inclusione: in particolare, se
- $F(\mathcal{C}) \subsetneq \mathrm{DATI}_\perp^{\mathrm{DATI}}$ allora esistono compiti **non automatizzabili**;
- $F(\mathcal{C}) = \mathrm{DATI}_\perp^{\mathrm{DATI}}$ allora l'informatica può fare tutto.

Se ci trovassimo nella prima situazione dovremmo individuare una sorta di "recinto" per dividere le funzioni calcolabili da quelle che non lo sono.

Calcolare una funzione vuole dire *risolvere un problema*: ad ognuno di questi associo una **funzione soluzione**, che mi permette di risolvere in modo automatico il problema.

Grazie a questa definizione, calcolare le funzioni equivale a risolvere problemi.

In che modo possiamo risolvere l'inclusione?

Un primo approccio è quello della **cardinalità**: viene definita come una funzione che associa ad ogni insieme il numero di elementi che contiene.

Sembra un ottimo approccio, ma presenta alcuni problemi: infatti, funziona solo quando l'insieme è finito, mentre è molto fragile quando si parla di insiemi infiniti.

Ad esempio, gli insiemi
- $\mathbb{N}$ dei numeri naturali e
- $\mathbb{P}$ dei numeri naturali pari

sono tali che $\mathbb{P} \subsetneq \mathbb{N}$ ma hanno cardinalità $|\mathbb{N}| = |\mathbb{P}| = \infty$.

Dobbiamo dare una definizione diversa di cardinalità, visto che possono esistere "*infiniti più fitti/densi di altri*", come abbiamo visto nell'esempio precedente.

# 3. Lezione 03

## 3.1. Relazioni di equivalenza

### 3.1.1. Definizione

Una **relazione binaria** $R$ su due insiemi $A, B$ è un sottoinsieme $R \subseteq A \times B$ di coppie ordinate. Una relazione particolare che ci interessa è $R \subseteq A^2$. Due elementi $a, b \in A$ sono in relazione $R$ se e solo se $(a, b) \in R$. Indichiamo la relazione tra due elementi tramite notazione infissa $aRb$.

Una classe molto importante di relazioni é quella delle **relazioni di equivalenza**: una relazione $R \subseteq A^2$ é una relazione di equivalenza se e solo se $R$ é $RST$, ovvero:

- **riflessiva**: $\forall a \in A \quad aRa$;
- **simmetrica**: $\forall a, b \in A \quad aRb \implies bRa$;
- **transitiva**: $\forall a, b, c \in A \quad aRb \land bRc \implies aRc$.

### 3.1.2. Partizione

Ad ogni relazione di equivalenza si può associare una **partizione**, ovvero un insieme di sottoinsiemi tali che:

- $\forall i \in \mathbb{N}^+ \quad A_i \neq \varnothing$;
- $\forall i, j \in \mathbb{N}^+ \quad i \neq j \implies A_i \cap A_j = \varnothing$;
- $\bigcup_{i \in \mathbb{N}^+} A_i = A$.

Diremo che $R$ definita su $A^2$ *induce* una partizione $A_1, A_2, \ldots$ su $A$.

### 3.1.3. Classi di equivalenza e insieme quoziente

Dato un elemento $a \in A$, la sua **classe di equivalenza** é l'insieme

$$[a]_R = \{b \in A \mid aRb\},$$

ovvero tutti gli elementi che sono in relazione con $a$, chiamato anche *rappresentante della classe.*

Si può dimostrare che:

- non esistono classi di equivalenza vuote $\rightarrow$ garantito dalla riflessività;
- dati $a, b \in A$ allora $[a]_R \cap [b]_R = \varnothing$ oppure $[a]_R = [b]_R \rightarrow$ in altre parole, due elementi o sono in relazione o non lo sono;
- $\bigcup_{a \in A} [a]_R = A$.

Notiamo che, per definizione, l'insieme delle classi di equivalenza é una partizione indotta dalla relazione $R$ sull'insieme $A$. Questa partizione é detta **insieme quoziente** di $A$ rispetto a $R$ ed é denotato con $A/R$.

## 3.2. Cardinalità

### 3.2.1. Isomorfismi

Due insiemi $A$ e $B$ sono **isomorfi** (*equinumerosi* o *insiemi che hanno la stessa cardinalità*) se esiste una biiezione tra essi. Formalmente scriviamo:

$$A \sim B.$$

Detto $\mathcal{U}$ l'insieme di tutti gli insiemi, la relazione $\sim$ é sottoinsieme di $\mathcal{U}^2$.

Dimostriamo che $\sim$ é una relazione di equivalenza:

- *riflessività*: $A \sim A$ se la biiezione é $i_A$;
- *simmetria*: $A \sim B \implies B \sim A$ se la biiezione é la funzione inversa;

- *transitività*: $A \sim B \land B \sim C \implies A \sim C$ se la biiezione é la composizione della funzione usata per $A \sim B$ con la funzione usata per $B \sim C$.

Dato che $\sim$ è una relazione di equivalenza, è possibile partizionare l'insieme $\mathcal{U}$. La partizione creata è formata da classi di equivalenza che contengono insiemi isomorfi, ossia con la stessa cardinalità.

Possiamo quindi definire la **cardinalità** come l'insieme quoziente di $\mathcal{U}$ rispetto alla relazione $\sim$.

Questo approccio permette di utilizzare la nozione di *cardinalità* anche con gli insiemi infiniti, dato che l'unica incognita da trovare è una funzione biettiva tra i due insiemi.

### 3.2.2. Cardinalità finita

La prima classe di cardinalità che vediamo é quella delle **cardinalità finite**. Prima di tutto definiamo la famiglia di insiemi:

$$J_n = \begin{cases} \varnothing & \text{se } n = 0 \\ \{1, ..., n\} & \text{se } n > 0 \end{cases}.$$

Un insieme $A$ ha cardinalità finita se $A \sim J_n$ per qualche $n \in \mathbb{N}$. In tal caso possiamo scrivere $|A| = n$.

La classe di equivalenza $[J_n]_\sim$ riunisce tutti gli insiemi di $\mathcal{U}$ contenenti $n$ elementi.

### 3.2.3. Cardinalità infinita

L'altra classe di cardinalità da studiare è quella delle **cardinalità infinite**, ovvero gli insiemi non in relazione con $J_n$.

#### 3.2.3.1. Insiemi numerabili

I primi insiemi a cardinalità infinita sono gli **insiemi numerabili**. Un insieme $A$ é numerabile se e solo se $A \sim \mathbb{N}$, ovvero $A \in [\mathbb{N}]_\sim$.

Gli insiemi numerabili sono "**listabili**", ovvero é possibile elencare *tutti* gli elementi dell'insieme $A$ tramite una regola, in questo caso la funzione $f$ biiezione tra $\mathbb{N}$ e $A$. Infatti, grazie alla funzione $f$, è possibile elencare gli elementi di $A$ formando l'insieme:

$$A = \{f(0),\ f(1),\ ...\}.$$

Questo insieme é esaustivo, quindi elenca ogni elemento dell'insieme $A$ senza perderne nessuno.

Gli insiemi numerabili più famosi sono:
- numeri pari $\mathbb{P}$ e numeri dispari $\mathbb{D}$;
- numeri interi $\mathbb{Z}$ generati con la biiezione $f(n) = (-1)^n \left( \frac{n + (n \bmod 2)}{2} \right)$;
- numeri razionali $\mathbb{Q}$.

Gli insiemi numerabili hanno cardinalità $\aleph_0$ (si legge *"aleph"*).

#### 3.2.3.2. Insiemi non numerabili

Gli **insiemi non numerabili** sono insiemi a cardinalità infinita che non sono listabili come gli insiemi numerabili, ovvero sono "più fitti" di $\mathbb{N}$.

Il *non poter listare gli elementi* si traduce in *qualunque lista generata mancherebbe di qualche elemento*, di conseguenza non sarebbe una lista esaustiva di tutti gli elementi.

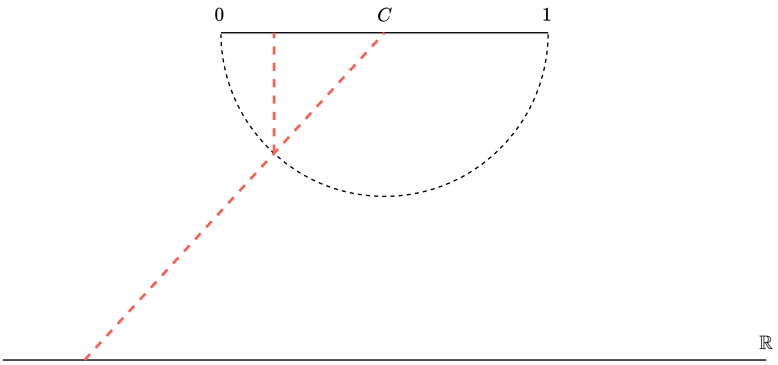
Il più famoso insieme non numerabile é l'insieme dei numeri reali $\mathbb{R}$.

**Teorema 3.2.3.2.1** L'insieme $\mathbb{R}$ non é numerabile

**Dimostrazione**
Suddividiamo la dimostrazione in tre punti:
1. dimostriamo che $\mathbb{R} \sim (0,1)$;
2. dimostriamo che $\mathbb{N} \nsim (0,1)$;
3. dimostriamo che $\mathbb{R} \nsim \mathbb{N}$.

[1] Partiamo con il dimostrare che $\mathbb{R} \sim (0,1)$: mostro che esiste una biiezione tra $\mathbb{R}$ e $(0,1)$. Usiamo una biiezione "grafica" costruita in questo modo:
- disegna la circonferenza di raggio $\frac{1}{2}$ centrata in $\frac{1}{2}$;
- disegna la perpendicolare al punto da mappare che interseca la circonferenza;
- disegna la retta passante per il centro $C$ e l'intersezione precedente.

L'intersezione tra l'asse reale e la retta finale é il punto mappato.




In realtà, $\mathbb{R}$ é isomorfo a qualsiasi segmento di lunghezza maggiore di 0.

La stessa biiezione vale anche sull'intervallo chiuso $[0,1]$ utilizzando la "compattificazione" $\mathring{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$ di $\mathbb{R}$, mappando 0 su $-\infty$ e 1 su $+\infty$.

[2] Continuiamo dimostrando che $\mathbb{N} \nsim (0,1)$: devo dimostrare che l'intervallo $(0,1)$ non é listabile, ovvero ogni lista che scrivo é un "colabrodo", termine tecnico che indica la possibilità di costruire un elemento che dovrebbe appartenere alla lista ma che invece non é presente.
Per assurdo sia $\mathbb{N} \sim (0,1)$, allora posso listare gli elementi di $(0,1)$ esaustivamente come:

$$\begin{aligned} &0.\ a_{00}\ a_{01}\ a_{02}\ \dots \\ &0.\ a_{10}\ a_{11}\ a_{12}\ \dots \\ &0.\ a_{20}\ a_{21}\ a_{22}\ \dots \\ &0.\ \dots \quad , \end{aligned}$$

dove con $a_{ij}$ indichiamo la cifra di posto $j$ dell'$i$-esimo elemento della lista.

Costruisco il *"numero colpevole"* $c = 0.c_0c_1c_2\dots$ tale che

$$c_i = \begin{cases} 2 \text{ se } a_{ii} \neq 2 \\ 3 \text{ se } a_{ii} = 3 \end{cases} .$$

In poche parole, questo numero é costruito "guardando" tutte le cifre sulla diagonale.

Questo numero sicuramente appartiene a $(0,1)$ ma non appare nella lista: infatti ogni cifra $c_i$ del colpevole differisce da qualunque numero nella lista in almeno una posizione, che é quella della diagonale. Ma questo é assurdo: avevamo assunto $(0,1)$ numerabile.

Quindi $N \nsim (0,1)$.

Questo tipo di dimostrazione é detta **dimostrazione per diagonalizzazione**.

[3] Terminiamo dimostrando che $\mathbb{R} \nsim \mathbb{N}$: per transitività. Vale il generico, ovvero non si riesce a listare nessun segmento di lunghezza maggiore di $0$. $\square$

L'insieme $\mathbb{R}$ viene detto **insieme continuo** e tutti gli insiemi isomorfi a $\mathbb{R}$ si dicono a loro volta *continui*. I più famosi insiemi continui sono:

- $\mathbb{R}$ insieme dei numeri reali;
- $\mathbb{C}$ insieme dei numeri complessi;
- $\mathbb{T} \subset \mathbb{I}$ insieme dei numeri trascendenti.

# 4. Lezione 04

## 4.1. Cardinalità

Vediamo due insiemi continui che saranno importanti successivamente.

### 4.1.1. Insieme delle parti

Il primo insieme che vediamo é l'**insieme delle parti**, o *power set*, di $\mathbb{N}$.

Quest'ultimo é l'insieme

$$P(\mathbb{N}) = 2^{\mathbb{N}} = \{S \mid S \text{ é sottoinsieme di } \mathbb{N}\}.$$

**Teorema 4.1.1.1** $P(\mathbb{N}) \nsim \mathbb{N}$.

**Dimostrazione**
Dimostriamo questo teorema con la diagonalizzazione.

Rappresentiamo il sottoinsieme $A \subseteq \mathbb{N}$ tramite il suo **vettore caratteristico**:

$$\begin{aligned} &\mathbb{N} : 0\ 1\ 2\ 3\ 4\ 5\ 6\ \ldots \\ &A : 0\ 1\ 1\ 0\ 1\ 1\ 0\ \ldots \quad . \end{aligned}$$

Il vettore caratteristico di un sottoinsieme é un vettore che nella posizione $p_i$ ha 1 se $i \in A$, altrimenti ha 0.

Per assurdo sia $P(\mathbb{N})$ numerabile. Vista questa proprietà posso listare tutti i vettori caratteristici che appartengono a $P(\mathbb{N})$ come

$$\begin{aligned} b_0 &= b_{00}\ b_{01}\ b_{02}\ \ldots \\ b_1 &= b_{10}\ b_{11}\ b_{12}\ \ldots \\ b_2 &= b_{20}\ b_{21}\ b_{22}\ \ldots \quad . \end{aligned}$$

Costruiamo un *colpevole among us* che appartiene a $P(\mathbb{N})$ ma non é presente nella lista precedente. Definiamo il vettore

$$c = \overline{b_{00}}\ \overline{b_{11}}\ \overline{b_{22}}\ldots$$

che contiene nella posizione $c_i$ il complemento di $b_{ii}$.

Questo vettore appartiene a $P(\mathbb{N})$ ma non é presente nella lista precedente perché é diverso da ogni elemento della lista in almeno una cifra.

Ma questo é assurdo perché $P(\mathbb{N})$ era numerabile, quindi $P(\mathbb{N}) \nsim \mathbb{N}$. □

Visto questo teorema possiamo affermare che:

$$P(\mathbb{N}) \sim [0,1] \sim \mathring{\mathbb{R}}.$$

### 4.1.2. Insieme delle funzioni

Il secondo insieme che vediamo é l'insieme delle funzioni da $\mathbb{N}$ in $\mathbb{N}$.

Quest'ultimo é l'insieme

$$\mathbb{N}_{\perp}^{\mathbb{N}} = \{f : \mathbb{N} \longrightarrow \mathbb{N}\}.$$

**Teorema 4.1.2.2** $\mathbb{N}_{\perp}^{\mathbb{N}} \nsim \mathbb{N}$.

**Dimostrazione**
Anche in questo caso useremo la dimostrazione per diagonalizzazione.

Per assurdo sia $\mathbb{N}_\perp^{\mathbb{N}}$ numerabile, quindi possiamo listare $\mathbb{N}_\perp^{\mathbb{N}}$ come $\{f_0, f_1, f_2, ...\}$.

| | 0 | 1 | 2 | 3 | ... | $\mathbb{N}$ |
|---|---|---|---|---|---|---|
| $f_0$ | $f_0(0)$ | $f_0(1)$ | $f_0(2)$ | $f_0(3)$ | ... | ... |
| $f_1$ | $f_1(0)$ | $f_1(1)$ | $f_1(2)$ | $f_1(3)$ | ... | ... |
| $f_2$ | $f_2(0)$ | $f_2(1)$ | $f_2(2)$ | $f_2(3)$ | ... | ... |
| $f_3$ | $f_3(0)$ | $f_3(1)$ | $f_3(2)$ | $f_3(3)$ | ... | ... |

Scriviamo un colpevole $\varphi : \mathbb{N} \longrightarrow \mathbb{N}_\perp$ per dimostrare l'assurdo. Una prima versione potrebbe essere la funzione $\varphi(n) = f_n(n) + 1$ per *disallineare* la diagonale, ma questo non va bene: infatti, se $f_n(n) = \perp$ non sappiamo dare un valore a $\varphi(n) = \perp + 1$.

Definiamo quindi la funzione

$$\varphi(n) = \begin{cases} 1 & \text{se } f_n(n) = \perp \\ f_n(n) + 1 & \text{se } f_n(n) \downarrow \end{cases}.$$

Questa funzione é una funzione che appartiene a $\mathbb{N}_\perp^{\mathbb{N}}$ ma non é presente nella lista precedente: infatti, $\forall k \in \mathbb{N}$ otteniamo

$$\varphi(k) = \begin{cases} 1 \neq f_k(k) = \perp & \text{se } f_k(k) = \perp \\ f_k(k) + 1 \neq f_k(k) & \text{se } f_k(k) \downarrow \end{cases}.$$

Ma questo é assurdo perché $P(\mathbb{N})$ era numerabile, quindi $P(\mathbb{N}) \nsim \mathbb{N}$. □

## 4.2. Potenza computazionale

### 4.2.1. Validità dell'inclusione $F(\mathcal{C}) \subseteq \text{DATI}_\perp^{\text{DATI}}$

Ora che abbiamo una definizione "potente" di cardinalità, essendo basata su strutture matematiche, possiamo verificare la validità dell'inclusione

$$F(\mathcal{C}) \subseteq \text{DATI}_\perp^{\text{DATI}}.$$

Diamo prima qualche considerazione:
- $\text{PROG} \sim \mathbb{N}$: identifico ogni programma con un numero, ad esempio la sua codifica in binario;
- $\text{DATI} \sim \mathbb{N}$: come prima, identifico ogni dato con la sua codifica in binario.

In poche parole, stiamo dicendo che programmi e dati non sono più dei numeri naturali $\mathbb{N}$.

Ma questo ci permette di dire che:

$$F(\mathcal{C}) \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_\perp^{\mathbb{N}} \sim \text{DATI}_\perp^{\text{DATI}}.$$

Questo é un risultato importantissimo: abbiamo appena dimostrato con la relazione precedente che **esistono funzioni non calcolabili**. Il problema é che *ho pochi programmi e troppe funzioni.*

Questo risultato però é arrivato considerando vere le due considerazioni precedenti: andiamo quindi a dimostrarle utilizzando le **tecniche di aritmetizzazione** (o *godelizzazione*) **di strutture**, ovvero delle tecniche che rappresentano delle strutture con un numero, così da avere la matematica e l'insiemi degli strumenti che ha a disposizione.

### **4.2.2.** DATI $\sim \mathbb{N}$

Vogliamo formare una legge che:

1. associ biunivocamente dati a numeri e viceversa;
2. consenta di operare direttamente sui numeri per operare sui corrispondenti dati, ovvero abbia delle primitive per lavorare il numero che "riflettano" il risultato sul dato senza ripassare per il dato stesso;
3. ci consenta di dire, senza perdita di generalità, che i nostri programmi lavorano sui numeri, ovvero grazie al punto 2 possiamo lavorare bene sui numeri senza ripassare dai dati.

**Teorema 4.2.2.1** $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$.

**Dimostrazione**

□

Estendiamo adesso il risultato all'interno insieme $\mathbb{N}$, ovvero

$$\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+ \rightsquigarrow \mathbb{N} \times \mathbb{N} \sim \mathbb{N}.$$

Grazie a questi risultati si può dimostrare che $\mathbb{Q} \sim \mathbb{N}$.

# 5. Lezione 05

I risultati ottenuti fino a questo punto ci permettono di dire che ogni dato è trasformabile in un numero, che può essere soggetto a trasformazioni e manipolazioni matematiche.

## 5.1. Strutture dati

Mostriamo come, tramite questo principio, è possibile mappare in numeri alcune delle principali strutture dati utilizzate nei programmi.

### 5.1.1. Liste

In generale, lavorando con delle liste non ne è nota la grandezza. Di conseguenza, ci serve sempre un modo per capire quando siamo arrivati all'ultimo elemento, in modo da sapere quando finiscono gli elementi della lista.

Codifichiamo la lista $x_1, ..., x_n$ con $< x_1, ..., x_n >$, in cui:

$$< x_1, ..., x_n >=< x_1, < x_2, < ... < x_n, 0 > ... \ggg .$$

Se volessimo codificare la lista $1, 2, 5$, otterremmo:

$$\begin{aligned} < 1, 2, 5 >&=< 1, < 2, < 5, 0 \ggg \\ &=< 1, < 2, 16 \gg \\ &=< 1, 188 > \\ &= 18144. \end{aligned}$$

Per decodificare una lista $M$, ci basterà utilizzare la funzione $\text{left}(M_i)$ fintanto che $\text{right}(M_n) \neq 0$. Siamo sicuri che non avremo problemi a salvare 0 nella lista in quanto non si confonderebbe con lo 0 che indica la fine della lista, perché uno appare nella parte sinistra, mentre l'altro nella parte destra.

Vogliamo anche che le funzioni Codifica e Decodifica siano implementabili facilmente. Assumiamo:

- che 0 codifichi la lista nulla;
- di avere routine per $<, >$, left e right.

Codifica

```
int encode(x_1, ..., x_n) {
  int k = 0;
  for (int i = n; i >= 1; i--)
    k = <x_1, k>
  return k;
}
```

Decodifica

```
void decode(n) {
    if (n != 0) {
      print(left(n));
      decode(rigt(n));
    }
}
```

Un'altra funzione utile è la funzione Lunghezza:

```
int length (int n) {
   return n == 0? 0 : 1 + length(right(n));
}
```

Definiamo anche la funzione Proiezione:

$$\text{proj}(t, n) = \begin{cases} -1 \text{ se } t > \text{length}(n) \parallel t = 0 \\ x_t \quad \text{altrimenti} \end{cases}$$

e la sua implementazione:

```
int proj(int t, int n) {
  if (t == 0 || t > length(n))
    return -1;
```

```
  else
    if (t == 1)
      return left(n);
    else
      return proj(t-1, right(n));
}
```

#### 5.1.2. Array

Per gli array, il discorso è più semplice, in quanto la dimensione è nota a priori. Di conseguenza, non necessitiamo di un carattere di fine sequenza. Dunque avremo che l'array $x_1, ..., x_n$ viene codificato in $[x_1, ..., x_n]$ dove:

$$[x_1, ..., x_n] = [x_1, ..., [x_{n-1}, x_n]...].$$

#### 5.1.3. Matrici

Discorso simile vale per una matrice, che codifica singolarmente le righe e successivamente codifica per tutte le colonne. La matrice $\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}$ viene codificata in: $\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = [[x_{11}, x_{12}], [x_{21}, x_{22}]]$.

#### 5.1.4. Grafi

Un primo modo per codificare i grafi è sfruttando le liste di adiacenza dei vertici. Consideriamo il seguente grafo: La sua codifica si ottiene da:

$$\ll 2, 3, 4 >, < 1, 2 >, < 1, 2, 4 >, < 1, 3 \gg = n$$

in cui, codifichiamo singolarmente ogni lista di adiacenza e successivamente codifichiamo i risultati del passo precedente.

Un secondo modo per farlo è sfruttando la matrice di adiacenza dei vertici. Una volta costruita la matrice, ci basta codificarla come abbiamo già descritto.

### 5.2. Applicazioni

Una volta visto come rappresentare le principali strutture dati, è facile trovare delle vie per codificare qualsiasi tipo di dato in un numero. Vediamo alcuni esempi.

#### 5.2.1. Testi

Dato un testo, possiamo ottenere un compressore in questo modo: trasformiamo il testo in numeri tramite le codifiche ASCII dei singoli caratteri e successivamente sfruttiamo l'idea dietro la codifica delle liste per codificare quanto ottenuto.

Per esempio, ciao $\rightarrow 9910597111 \rightarrow < 99, < 105, < 97, < 111, 0 \ggg$ .

*Perché non è un buon compressore?*
Si vede facilmente come i bit necessari a rappresentare il numero associato al testo crescano esponenzialmente sulla lunghezza dell'input. Ne segue che questo è un *pessimo* modo per comprimere messaggi.

*Perché non è un buon sistema crittografico?*
La natura stessa del processo garantisce la possibilità di trovare un modo per decrittare in modo analitico, di conseguenza chiunque potrebbe in poco tempo decifrare il mio messaggio. Inolte è molto sensibile agli errori.

#### 5.2.2. Suoni

Dato un suono, possiamo campionare il suo impulso elettrico e codificare i valori campionati. Diventa una codifica di un array di numeri.

#### 5.2.3. Immagini

Esistono diverse tecniche, per esempio la **bitmap**: ogni pixel contiene la codifica numerica di un colore $\Rightarrow$ codifico separatamente ogni pixel e infine codifico insieme i risultati appena ottenuti.

## 5.3. Conclusioni

Abbiamo mostrato come i dati possano essere buttati via, per considerare solo i numeri associati ad essi.

Di conseguenza, possiamo sostituire tutte le funzioni $f : \mathrm{DATI} \to \mathrm{DATI}_\perp$ con delle funzioni $f' : \mathbb{N} \to \mathbb{N}$.

In altre parole, l'universo dei problemi per i quali cerchiamo una soluzione automatica è rappresentabile da $\mathbb{N}_\perp^\mathbb{N}$.