Algoritmi e Strutture Dati

Lezione 2 | 24 Febbraio

Università degli Studi di Modena e Reggio Emilia — UnimoreE

1. Più algoritmi per lo stesso problema

Definizione

Lo stesso problema può avere **algoritmi diversi**: se sono tutti corretti, producono sempre lo stesso output dallo stesso input, ma possono funzionare in modo completamente diverso e usare un numero diverso di operazioni.

1.1. Esempio 1 — Trovare il massimo tra n elementi

Data la sequenza:

24 15 7 9 20 49 33 35 22 40 52 1 62 30 8 43

Il computer non vede la sequenza tutta in una volta: la legge **un elemento alla volta**, come se avesse una "benda sugli occhi".

1.1.1. Algoritmo 1 — Scansione lineare

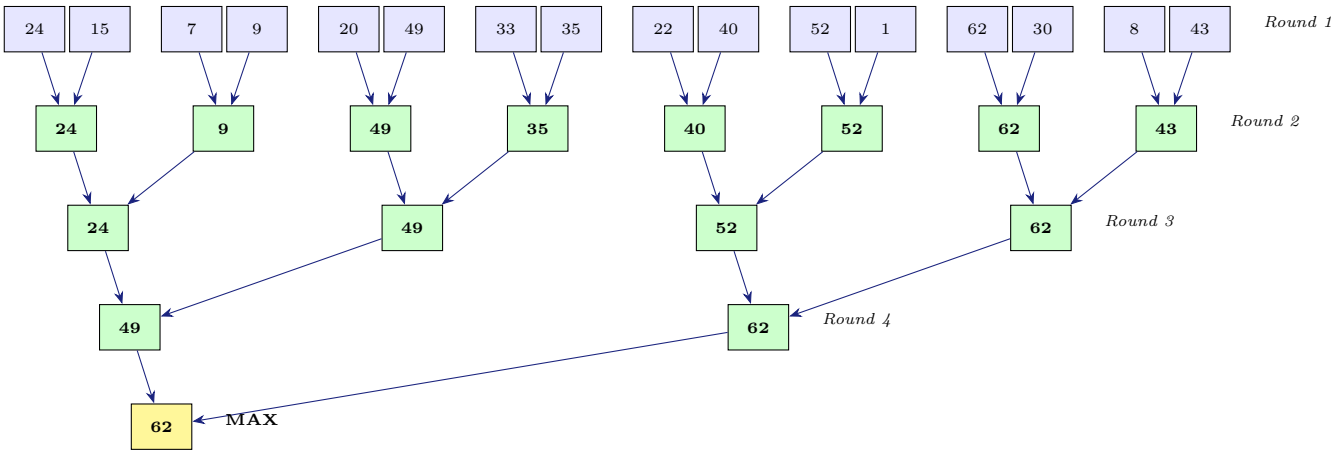
1. Prendi il primo elemento e consideralo il massimo temporaneo.
2. Scorri la sequenza uno a uno: se il numero corrente è maggiore del massimo corrente, aggiornalo.
3. Alla fine della scansione, il massimo temporaneo è il massimo assoluto.

```
max = 24
  -> 15 < 24  -> max rimane 24
  ->  7 < 24  -> max rimane 24
  -> 49 > 24  -> max = 49
  -> ...
  -> 62 > 52  -> max = 62
  -> fine: MAX = 62
```

Listing 1: Scansione lineare — traccia di esecuzione

1.1.2. Algoritmo 2 — Tecnica a torneo

Si confrontano i numeri a **coppie**, come in un torneo sportivo. I vincitori si sfidano tra loro sino a quando non rimane un solo vincitore.



Nota

Se n non è una potenza di 2, il numero rimasto "da solo" gareggia alla fine contro il vincitore corrente. Se due numeri sono uguali, non importa quale passa avanti.

1.1.3. Confronto tra i due algoritmi

	Algoritmo 1 (lineare)	**Algoritmo 2 (torneo)**
Idea	Scansione uno a uno	Confronti a coppie ricorsivi
Correttezza	✓	✓
Risultato	MAX = 62	MAX = 62
Operazioni (n elem.)	$n-1$ confronti	$n-1$ confronti

I due algoritmi sono **equivalenti**: idee diversissime, ma stesso numero di operazioni. Vedremo più avanti perché $n-1$ è il limite inferiore del problema.

1.2. Esempio 2 — Massimo Comune Divisore (MCD)

Input: due numeri naturali m e n **Output:** $\gcd(m, n)$

1.2.1. Algoritmo 1 — Per intersezione di insiemi

$$I = \{d \in \mathbb{N} \mid d \text{ divide } m\}, \quad J = \{d \in \mathbb{N} \mid d \text{ divide } n\}$$

$$K = I \cap J \quad \text{(divisori comuni)}, \quad d = \max(K) \quad \Rightarrow \quad \text{restituisci } d$$

1.2.2. Algoritmo 2 — Algoritmo di Euclide

```
while m != n do:
    if m > n  then  m := m - n
    else            n := n - m
return n
```

Listing 2: Algoritmo di Euclide

Entrambi sono corretti, ma usano un **numero diverso di operazioni**. L'algoritmo di Euclide è molto più rapido per numeri grandi.

2. Categorie di algoritmi

Il concetto tradizionale di algoritmo si è evoluto. Esistono oggi diverse accezioni:

Tipo	Descrizione	Esempio
Non deterministici / randomizzati	Esecuzioni diverse dallo stesso input possono dare output diversi — si fa una scelta casuale deliberata	Algoritmi probabilistici
Non terminanti	Pensati per processi che non si fermano mai	Sistema operativo
On-line	L'input non è disponibile tutto subito, ma arriva durante l'esecuzione	Previsioni di borsa
Paralleli e distribuiti	Eseguiti su più macchine (o più core) che comunicano tra loro	Sistemi distribuiti
Quantistici	Lavorano su principi fisici completamente diversi	(fuori dal nostro scope)

Nota

I processori moderni hanno più core: questo rende la programmazione parallela comune ma complessa. I processi devono coordinarsi, altrimenti si danno fastidio. La memoria condivisa vs. separata è uno dei problemi principali (argomento della magistrale).

3. Come si impara a progettare algoritmi?

Non esiste un "bottone magico" in cui si inserisce un problema e viene fuori la soluzione. Bisogna costruirsi la mentalità giusta su due fronti:

3.1. Studio

- I problemi noti, già risolti, le loro soluzioni, i costi, i limiti
- Le **tecniche standard** (torneremo spesso su questo)
- I **limiti teorici**: alcune cose non si possono fare meglio di un certo threshold — qualcuno lo ha già dimostrato. Si prende atto del limite e si va avanti.

3.2. Esercizio

- Sviluppare la capacità di affrontare **problemi nuovi** in autonomia
- Risolvere su carta è più difficile che programmare (niente autocomplete, niente debugging automatico) — ma fa migliorare di più

Attenzione

Prima di scrivere codice, **pensa** all'algoritmo. Progettare prima di programmare è quasi sempre la scelta giusta.

4. Proprietà 1 — Correttezza

Correttezza

Per ogni **istanza di input legale**, l'algoritmo deve **terminare** e restituire l'**output corrispondente**.

Come ci si convince che un algoritmo sia corretto?

× Provarlo su 2–3 casi non basta

× Non possiamo provare tutti gli input (spesso infiniti)

✓ L'unico modo è una **dimostrazione matematica**

Teorema

Basta un solo controesempio per dimostrare che un algoritmo è sbagliato. Non serve trovare tutti i casi in cui fallisce — uno solo è sufficiente.

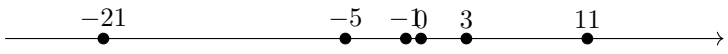
4.1. Esempio pratico — Il problema del tour minimo

Input: n punti sul piano **Output:** percorso che visiti tutti i punti esattamente una volta, tornando al punto di partenza, nel modo più breve possibile.

4.1.1. Idea 1 — Nearest Neighbour (vicino più vicino)

```
1. Scegli un punto di partenza
2. Finche' ci sono punti non visitati:
       vai al punto non visitato piu' vicino alla posizione attuale
3. Torna al punto di partenza
```

Consideriamo questi punti su una retta:



Partendo da 0, il nearest neighbour produce:

$$0 \to -1 \to 1 \to 3 \to 11 \to -5 \to -21 \qquad \text{costo} \approx 82$$

Ma esiste un tour più corto:

$$0 \to 3 \to 11 \to \ldots \to -5 \to -21 \qquad \text{costo} \approx 54$$

$54 < 82 \Rightarrow$ l'algoritmo **non è ottimo**. ×

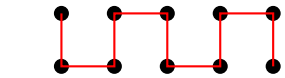
Nota

Non devo dimostrare che 54 sia il minimo assoluto: mi basta mostrare che esiste qualcosa di meglio di 82.

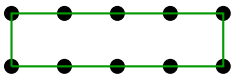
4.1.2. Idea 2 — Connetti la coppia più vicina

```
Ripetutamente collega la coppia di punti non ancora connessa con
   distanza minore,
senza creare cicli e senza visitare un punto piu' di due volte.
```

Su una certa configurazione di punti, l'algoritmo produce:



tour prodotto dall'algoritmo



tour più corto

Anche qui un controesempio è sufficiente. ×

4.1.3. Idea 3 — Forza bruta (tutti i tour possibili)

```
1. Genera tutti gli ordinamenti possibili degli n punti
2. Per ogni ordinamento, calcola la lunghezza del tour
3. Restituisci il tour piu' corto
```

È corretto? Sì — si provano tutte le possibilità, quindi si trova sicuramente il minimo.

Ma è efficiente? Il numero di ordinamenti di n punti è $n!$:

$$4! = 24 \qquad 10! = 3\,628\,800 \qquad 80! \approx 7 \times 10^{118}$$

Attenzione

Un algoritmo corretto ma inapplicabile nella pratica **non è una soluzione accettabile**.

Spoiler: ad oggi non si sa se esista una soluzione ottima ed efficiente per questo tipo di problema. È la questione aperta **P vs NP**.

5. Proprietà 2 — Efficienza

Efficienza

Per restituire la soluzione, l'algoritmo deve utilizzare il **minimo delle risorse necessarie**. Le risorse principali sono **tempo** e **memoria**. Il corso si concentra soprattutto sul tempo.

5.1. Perché non usare l'orologio?

Il tempo in secondi dipende da:

- La **macchina** su cui gira (processore più veloce = meno secondi)
- L'**istanza** di input (4 nodi ≈ 1 sec, 10^9 nodi ≈ 50 anni)

Vogliamo una misura **indipendente dalla macchina** che permetta di **confrontare algoritmi**.

5.2. La soluzione: contare le operazioni elementari

Definizione

TEMPO $\longrightarrow$ Numero di istruzioni elementari da eseguire

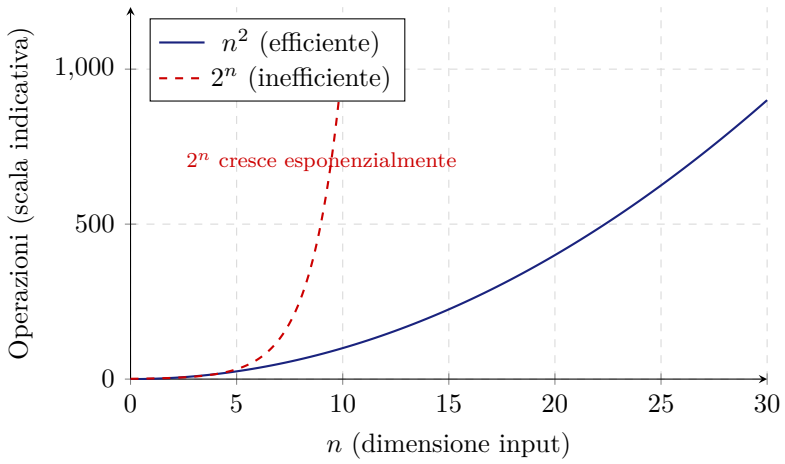
Analogia: esprimere la distanza tra città in **km** (fissi) invece che in ore (dipendenti dalla velocità).

5.3. E se usassi un supercomputer?

Attenzione

Un algoritmo più efficiente eseguito su un computer **lento** terminerà **sempre prima** di uno inefficiente su un computer **veloce**, per un numero sufficientemente grande di input.

	Operazioni	**Macchina**	**Tempo di esecuzione**	
			$n = 30$	$n = 60$
Algoritmo A (efficiente)	n^2	lenta: 10^5 op/s	0.009 s	0.036 s
Algoritmo B (inefficiente)	2^n	veloce: 10^7 op/s	17.9 min	$\approx 3.66 \times 10^{11}$ anni



6. Analisi del tempo: il caso peggiore

Caso peggiore

Noi misuriamo sempre il tempo nel **caso peggiore**: l'istanza che richiede il **massimo numero di operazioni** per produrre il risultato.

Il caso peggiore fornisce un **limite superiore** garantito: se si attende quel numero di operazioni, l'algoritmo **sicuramente termina**, indipendentemente dall'istanza.

6.1. Esempio — Trovare il massimo (ricapitolo)

Con n elementi, entrambi gli algoritmi visti fanno esattamente $n - 1$ confronti.

Si può fare meglio di $n - 1$?

Limite inferiore del problema

Usando solo confronti, **non è possibile trovare il massimo di n elementi con meno di $n-1$ confronti.**
Dimostrazione:

- Devo stabilire un "vincitore" tra n elementi.
- "Vincitore" implica $n-1$ "sconfitti", ognuno dei quali ha perso *almeno* un confronto.
- Da ogni confronto esce *uno* sconfitto.
- Servono quindi *almeno* $n-1$ confronti. □

I due algoritmi trovati sono quindi **ottimi**: usano il menor numero possibile di operazioni. ✓

6.2. Esempio — Ricerca lineare in un array

Input: sequenza di n interi $A = A[0], A[1], \ldots, A[n-1]$ e un numero x
Output: la posizione di x in A, oppure "non trovato"

```
Search(A, x):
    i := 0
    while i < n and A[i] != x do
        i := i + 1

    if i = n then
        return "x non c'e'"
    else
        return "x sta in posizione i"
```

Listing 3: Ricerca lineare

Caso	Operazioni
Migliore	$x = A[0]$: **1 confronto**
Peggiore	$x \notin A$: n **confronti** (tutti gli elementi esaminati)

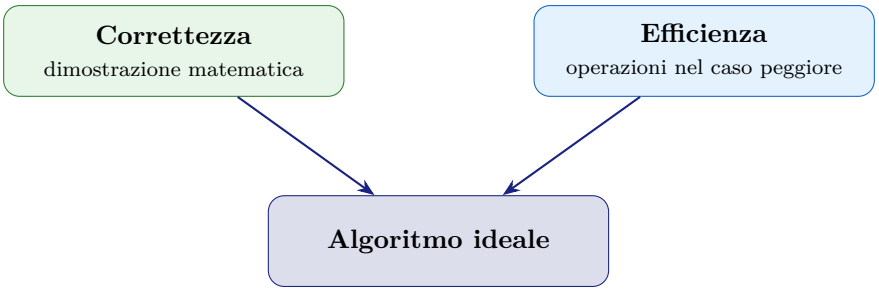
$$T(n) = n \quad \text{confronti nel caso peggiore}$$

Nota

Il caso peggiore dipende *da come è fatta l'istanza* (presenza o assenza di x), non solo dalla sua dimensione. Ma indipendentemente dall'istanza, non si faranno mai più di n confronti — questo è il limite superiore garantito.

7. Riepilogo: le due proprietà fondamentali

Proprietà	Descrizione
Correttezza	Per ogni istanza di input legale, l'algoritmo termina e restituisce l'output atteso. Si dimostra matematicamente: un solo controesempio è sufficiente a falsificarla.
Efficienza	L'algoritmo usa il minimo delle risorse necessarie. Si misura contando le operazioni elementari nel **caso peggiore**, indipendentemente dalla macchina.


Correttezza
dimostrazione matematica
Efficienza
operazioni nel caso peggiore
Algoritmo ideale