Algoritmi e Strutture Dati

Lezione 3 | 25 Febbraio

Università degli Studi di Modena e Reggio Emilia — Unimore

1. Efficienza: Tempo e Spazio

Gli algoritmi devono essere efficienti rispetto a due risorse:

Risorsa	Caratteristica
Tempo	Non è riutilizzabile — una volta usato, non si riavvolge.
Spazio	È riutilizzabile: la memoria può essere sovrascritta.

Per questo il tempo ci interessa *di più* come metrica. Ma la memoria non è illimitata (e la RAM costa!), e ci sono tre ragioni per non ignorarla:

1. Ha un **costo** economico.
2. Accedere alla memoria **secondaria** (disco) richiede molto più tempo di quella principale.
3. Potrebbe non essere **sufficiente** comunque (BigData, sistemi embedded).

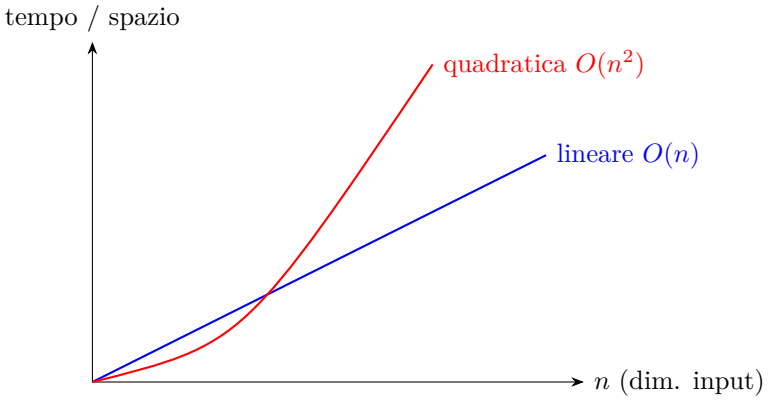
Nota

Ciò che misuriamo non è lo spazio totale usato, ma lo spazio **aggiuntivo** rispetto a quello necessario per memorizzare l'input. Scrivere l'input è inevitabile — ci interessa quanto extra serve.

2. Come si misura l'efficienza

Non ci interessa il conto preciso, ma l'**ordine di grandezza**: come crescono tempo e spazio al crescere dell'input?

$$f(n) = O(n)?\ O(n^2)?\ O(2^n)?\ O(\log n)?$$



Ci interessa capire il **comportamento all'infinito** — non il valore esatto per un input specifico. Man mano che n cresce, il tempo (o lo spazio) cresce piano o velocissimo?

3. Dimensione dell'input

Prima di misurare il tempo, dobbiamo definire **cosa intendiamo per "quanto è grande l'input"**.

Dimensione dell'input

Quanto tempo / spazio serve per *scrivere* l'input stesso. Tutto il resto (tempo/spazio dell'algoritmo) è in aggiunta.

3.1. Criterio di costo uniforme

Def. — Criterio di costo uniforme

La dimensione dell'input è il **numero di elementi** che lo costituiscono.

Input	Dimensione
n numeri	n
n coppie	n
Grafo con n nodi e m archi	$n + m$

Questo è il criterio che useremo **quasi sempre**: semplice e ottimo per la maggior parte dei casi.

3.2. Criterio di costo logaritmico

Def. — Criterio di costo logaritmico

La dimensione dell'input è il **numero di bit** necessari per rappresentarlo.

Esempio: trovare i fattori primi di un numero.

- Input $= 10 \rightarrow$ pochi bit
- Input $= 110012 \rightarrow$ più bit
- Input $= 10^9 \rightarrow$ molti bit

Non posso contare solo "quanti numeri ho" — un numero grande *è fisicamente più grande*. Per rappresentare un numero naturale n in binario (base 2) servono:

$$\lceil \log_2 n \rceil \quad \text{bit}$$

Nota

Useremo sempre logaritmi in **base 2**. Il criterio logaritmico è il più corretto teoricamente ma il criterio uniforme è il nostro standard — quello logaritmico compare solo dove necessario.

4. Progettare algoritmi

4.1. Step 1 — Capire il problema

Prima di tutto: **capire cosa bisogna fare**. Ovvio, eppure spesso sottovalutato.

4.2. Step 2 — Scegliere un formalismo

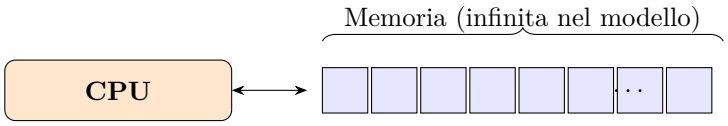
Il linguaggio naturale è ambiguo per natura — non va bene per algoritmi precisi.

Formalismo	Pro	Contro
Linguaggio naturale	Facile da scrivere	Ambiguo, impreciso
Diagrammi di flusso	Visuale, intuitivo	Scomodo per algoritmi grandi
Pseudo-codice ←	Flessibile, semi-formale	—
Linguaggio vero	Eseguibile	Troppo dettagliato, distrae

Lo pseudo-codice è il nostro strumento: abbastanza formale da essere preciso, abbastanza flessibile da permettere qualche parola in italiano senza ambiguità. Ci svincola da un preciso linguaggio di programmazione e ci fa concentrare sulla *soluzione* (il *cosa* e il *come*), non sui dettagli di implementazione.

5. Il Modello RAM (Random-Access-Memory)

Usiamo un modello ideale di computer per ragionare sull'efficienza:



Caratteristiche del modello RAM

- **Memoria infinita** — nel modello assumiamo di avere tutta quella che serve.
- **Accesso uniforme** — il tempo di lettura/scrittura è 1 unità per ogni cella, indipendentemente dalla posizione.
- **Singolo processore** — un'unica operazione alla volta. In un'unità di tempo: lettura *oppure* esecuzione istruzione *oppure* scrittura.
- **Istruzioni ammesse**: operazioni logico-aritmetiche, assegnamento, accesso tramite puntatore.

6. Pseudo-codice: la sintassi

Lo pseudo-codice **non è eseguibile** direttamente. Ci permette di descrivere un algoritmo senza occuparci di dettagli di basso livello.

6.1. Funzione vs Procedura

Tipo	Restituisce?	Quando
Funzione	Sì (`return val`)	Quando serve un risultato
Procedura	No (`return`)	Quando l'effetto è sul dato stesso

Esempio di funzione:

```
indice-massimo(A)
    n := length(A)
    max := 0
    for i = 1 to n-1 do    // ATTENZIONE: l'estremo e' INCLUSO (
       diverso da Python)
        if A[i] > A[max]
            then max := i
    return max
```

Esempio di procedura:

```
scambia(A, i, j)
    tmp := A[i]
    A[i] := A[j]
    A[j] := tmp
    return
```

Procedura che invoca funzioni e procedure:

```
scambio-massimo(A, i)
    j := indice-massimo(A)    // si sospende, aspetta il risultato,
       poi riparte
    if i != j
        then scambia(A, i, j) // si sospende, aspetta scambia, poi
           termina
```

Nota

Quando si chiama una funzione, l'esecuzione corrente si **sospende** e aspetta il risultato. Questo può succedere a cascata: funzione → funzione → funzione → ... Tutto si risolve a catena quando la funzione più interna termina.

7. Istruzioni principali dello pseudo-codice

Istruzione	Sintassi	Note
Assegnamento	`x := expr`	
Condizionale	`if P then A`	
	`if P then A else B`	
While	`while P do A`	
Repeat-until	`repeat A until P`	Ripete finché P diventa vera
For (crescente)	`for i = a to b [step k] do`	Estremi **inclusi**
For (decrescente)	`for i = a downto b [step k] do`	Estremi **inclusi**
Terminazione	`return`, `return(val)`	O sempre o mai
Errore	`error`	
Commento	`// testo`	

VIETATO: Break e Continue

Usare `break` o `continue` in una soluzione d'esame equivale a **punteggio zero**. Rendono il codice più corto ma molto più difficile da analizzare, dimostrare corretto e capire.

8. Strutture Dati

Definizione formale

"Insieme di dati **indirizzabile con un solo nome**, organizzato in modo specifico."

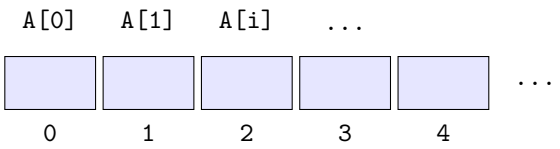
"Indirizzabile con un solo nome" = posso identificare tutta la struttura con un unico nome (es. `A` per un array) e accedere ai dati in modo predicibile.

Usare strutture dati permette di progettare algoritmi più **semplici, eleganti ed efficienti**. A volte il problema intero si riduce a scegliere la struttura dati giusta.

8.1. Strutture Elementari

Strutture il cui funzionamento **rispecchia direttamente** come i dati sono salvati in memoria fisicamente.

Esempio — Array:



Quando scrivo `A[i]`, vado direttamente alla cella di memoria `A + i`. Il modello è identico alla struttura fisica — nessuna discrepanza.

Altre strutture elementari: **Liste**, **Alberi**, **Grafi**.

8.2. Strutture Astratte (ADT — Abstract Data Type)

Si discostano da come il dato è salvato in memoria. Definiscono:

- Le **proprietà** dell'insieme di dati da memorizzare.
- Le **operazioni primitive** ammesse.

Non dicono *come* vengono implementate internamente.

Analogia perfetta: la pedaliera di un'auto

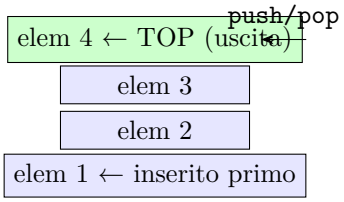
Sai che hai frizione, freno, acceleratore. Sai le proprietà (freno e acceleratore non si premono insieme). Sai le operazioni (acceleratore = vai più veloce, freno = rallenta). Non sai — e non ti interessa — com'è realizzata la meccanica sotto.

Attenzione

La stessa struttura astratta può avere **implementazioni diverse** con prestazioni diverse. Alcune operazioni possono essere più efficienti in un'implementazione rispetto a un'altra — è importante scegliere quella giusta per il problema.

9. Esempio ADT: la Pila (Stack)

La **Pila** è una struttura dati con comportamento **LIFO** (Last In, First Out).



La pila cresce verso l'alto; si accede solo dalla cima.

Operazioni primitive:

Operazione	Descrizione
`stack_init()`	Inizializza una pila vuota
`empty(S)`	Restituisce `true` se e solo se la pila è vuota
`push(S, val)`	Inserisce `val` in cima alla pila
`pop(S)`	Restituisce e **rimuove** l'ultimo valore inserito
`top(S)`	Restituisce l'ultimo valore inserito, **senza** rimuoverlo

Nota

Queste operazioni *descrivono* la Pila come ADT. Non dicono nulla su come viene implementata in memoria (potrebbe essere un array, una lista concatenata, ecc.). Ogni implementazione ha le sue caratteristiche di efficienza.

10. Nota sulle implementazioni: Cormen vs Bertossi/Montresor

Due approcci nei libri:

Stile	Linguaggio	Esempio
Cormen ← usiamo questo	Procedurale	`pop(S)`
Bertossi/Montresor	A oggetti	`S.pop()`

Attenzione

Noi usiamo l'approccio **Cormen** (procedurale): le primitive sono funzioni/procedure che prendono la struttura dati come parametro. Quando vedete `pop(S)` leggete "chiama pop sulla struttura `S`".