Complessità Computazionale degli Algoritmi

La "Dimensione" dei Problemi Conta!

Immaginate di cercare una parola specifica in un libro:

- In un libro di **10 pagine**, è rapido.
- In un libro di **100 pagine**, ci vuole più tempo.
- In un'**enciclopedia**, può diventare frustrante!

La **dimensione del problema** (il numero di pagine) influenza direttamente il tempo che impieghiamo. Lo stesso accade con gli algoritmi e la quantità di dati che devono elaborare.

Cos'è la complessità computazionale?

La **complessità computazionale** misura le RISORSE di calcolo necessarie per l'esecuzione di un algoritmo.

Ma cosa si intende per risorse?

- Spazio di memoria

- Tempo di esecuzione

In genere ci si concentra sulla complessità temporale.

A cosa serve la complessità computazionale?

- **Scegliere l'algoritmo giusto:** Per problemi con grandi quantità di dati, un algoritmo con complessità bassa (come O(log n)) sarà enormemente più veloce di uno con complessità alta (come $O(n^2)$).
- **Prevedere le prestazioni:** Ci aiuta a capire se il nostro programma sarà efficiente anche con tanti utenti o molti dati.
- **Ottimizzare il codice:** Se il programma è lento, analizzare la complessità ci indica dove concentrare i nostri sforzi per migliorarlo.
- **Scalabilità:** Un buon algoritmo con bassa complessità permette al nostro software di "scalare", cioè di funzionare bene anche quando il carico di lavoro aumenta.

Perché è importante?

Capire la complessità computazionale è una competenza cruciale per ogni informatico.

Ci permette di scrivere codice non solo funzionante, ma anche efficiente e scalabile.

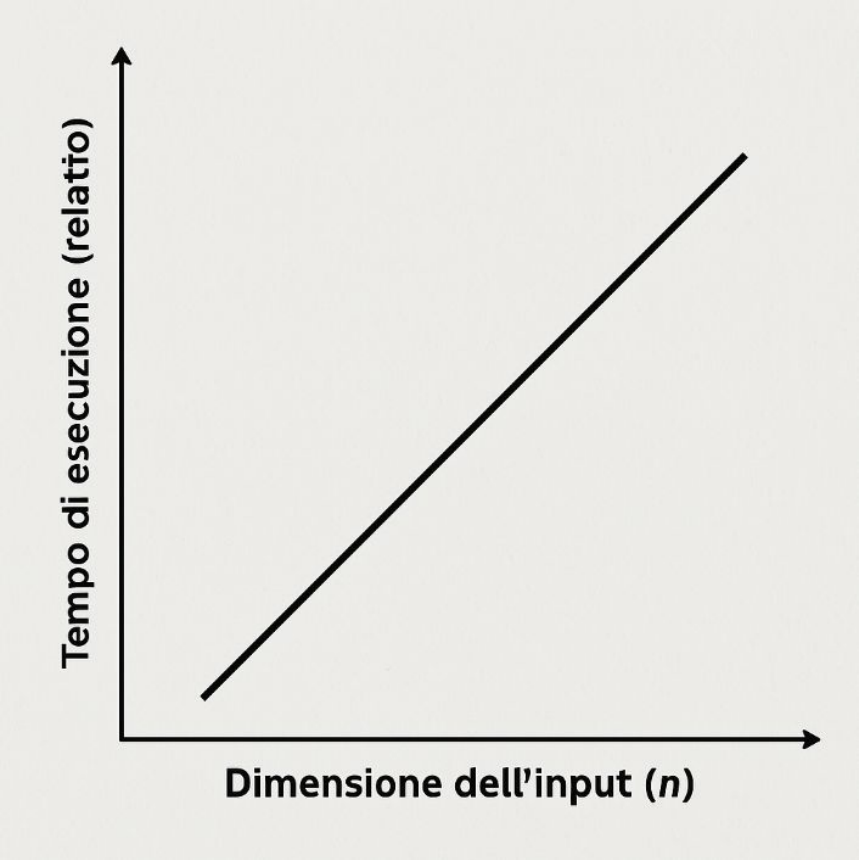
È un modo per far lavorare il computer in modo "intelligente", soprattutto quando i problemi diventano grandi e complessi.

Notazione "O Grande": Un Modo Semplice per Esprimere la Crescita

Usiamo la **notazione "O Grande"** per descrivere questa crescita. È un modo per dire "nel caso peggiore, il tempo di esecuzione crescerà all'incirca come questa funzione della dimensione dell'input (n)"

Esempi di complessità

- O(n): lineare, es. scorrere una lista
- O(1): costante, es. accedere a un elemento di un array
- O(log n): logaritmica, es. ricerca binaria
- $O(n^2)$: quadratica, es. algoritmi con due cicli annidati



Esempio 1: Ricerca Lineare - O(n)

Problema: Cercare un numero in una lista non ordinata.

Algoritmo: Controllare ogni numero uno dopo l'altro finché non lo troviamo (o arriviamo alla fine).

Funzione di costo: Se la lista ha n elementi, nel caso peggiore dobbiamo controllare tutti gli n elementi.

Quindi la complessità è O(n).

Esempio 2: Accesso ad un Elemento di un Array - O(1)

Problema: Accedere a un elemento specifico di un array sapendo la sua posizione (indice).

Algoritmo: Il computer va direttamente alla posizione di memoria indicata.

Funzione di costo: Non importa quanto sia grande l'array, ci vuole sempre lo stesso tempo per accedere a un elemento tramite il suo indice.

La complessità è O(1) (tempo costante).

Esempio 3: Ricerca Binaria - O(log n)

Problema: Cercare un numero in una lista **ordinata**.

Algoritmo: Si confronta l'elemento cercato con l'elemento centrale. Se non è quello giusto, si "dimezza" la parte della lista in cui cercare. Si ripete finché non si trova l'elemento o la parte da cercare diventa vuota.

Funzione di costo: Ad ogni passo, la dimensione del problema si dimezza. Il numero di volte che possiamo dimezzare una lista di n elementi prima di arrivare a 1 è circa log2(n).

La complessità è **O(log n)** (tempo logaritmico).

Esempio 4: Confronto a Coppie - $O(n^2)$

Problema: Confrontare ogni elemento di una lista con tutti gli altri elementi.

Algoritmo: Usiamo due cicli annidati. Il ciclo esterno scorre ogni elemento, e il ciclo interno lo confronta con tutti gli altri.

Funzione di costo: Se la lista ha n elementi, il ciclo esterno gira n volte, e per ogni volta il ciclo interno gira n volte. Quindi, il numero totale di operazioni è circa $n \times n = n2$.

La complessità è **$O(n^2)$** (tempo quadratico).

Grafico con esempi di costo