

# **Big O: L'Arte di Misurare l'Efficienza del Codice**

Una guida pratica per trasformare la  
teoria in performance.

# Il Dilemma dello Sviluppatore

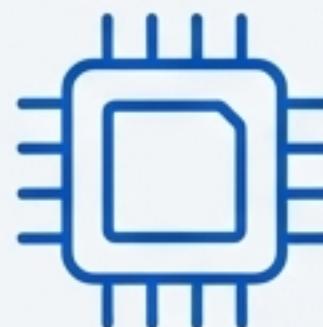
Dato un array di numeri, `nums`, come troviamo il valore più grande?

```
// Trova il massimo in un array 'nums' di lunghezza n
massimo = nums[0]
per ogni numero in nums:
    se numero > massimo:
        massimo = numero
```

**\*\*Questo codice è ‘efficiente’? Come facciamo a sapere se un’altra soluzione è migliore?\*\***

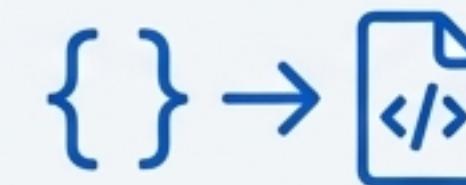
# Perché "Più Veloce" non è la risposta giusta

Il primo istinto potrebbe essere quello di misurare il tempo di esecuzione. Ma questo approccio è ingannevole.



## Hardware Diverso

Il tuo laptop non è un server di produzione.



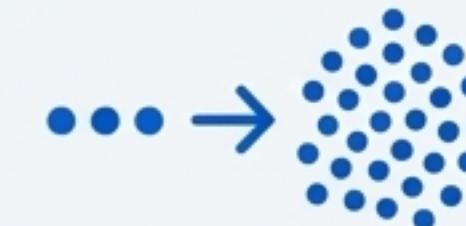
## Linguaggio e Compilatore

Python non ha le stesse performance di C++.



## Carico del Sistema

Altri processi in esecuzione possono falsare i risultati.



## Input di Test

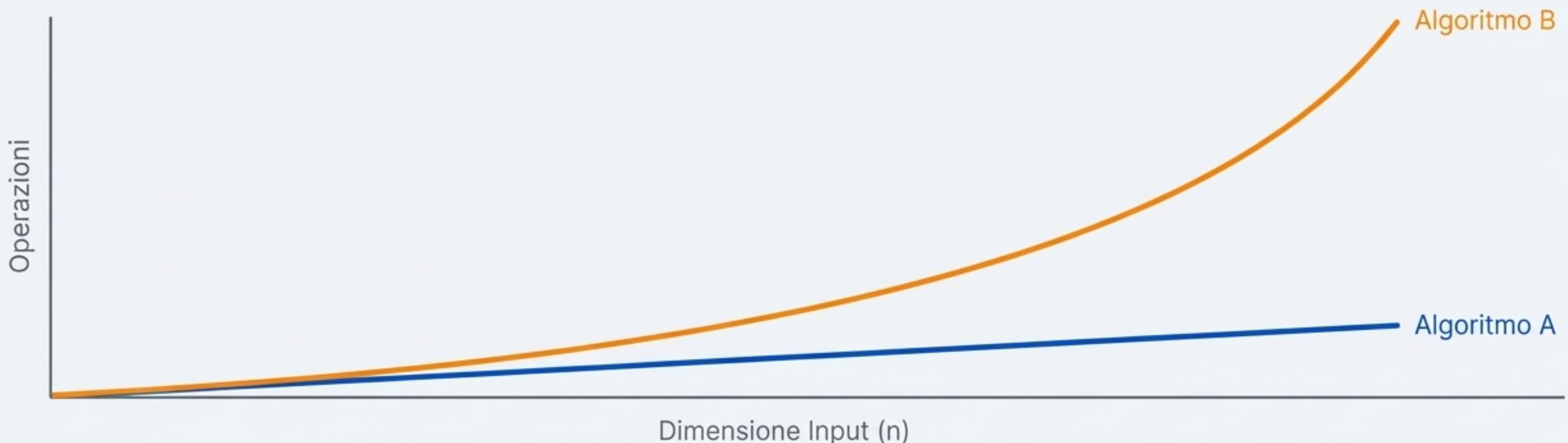
Funziona velocemente con 10 elementi, ma con 10 milioni?

Abbiamo bisogno di un modo per analizzare l'efficienza che sia **universale** e **indipendente dalla macchina**.

La Vera Domanda non è “Quanto è veloce?”, ma...

# Come si comporta l'algoritmo al crescere della dimensione dell'input?

Questo concetto è chiamato **Complessità Temporale**: Al crescere della dimensione dell'input, quanto tempo in più impiega l'algoritmo per essere completato?



# La Soluzione: Una Lingua Comune per la Complessità

La **Notazione Big O** descrive la relazione tra la **dimensione dell'input (n)** e le **risorse** (tempo o memoria) richieste da un algoritmo.



# I Due Pilastri dell'Efficienza



## Complessità Temporale (Time Complexity)

Quanto cresce il numero di operazioni al crescere dell'input?

Focus:  
CPU, calcoli, cicli.



## Complessità Spaziale (Space Complexity)

Quanta memoria aggiuntiva viene allocata al crescere dell'input?

Focus:  
Variabili, strutture dati, copie di array.

# Le Regole del Gioco (1/2): L'Andamento è Tutto

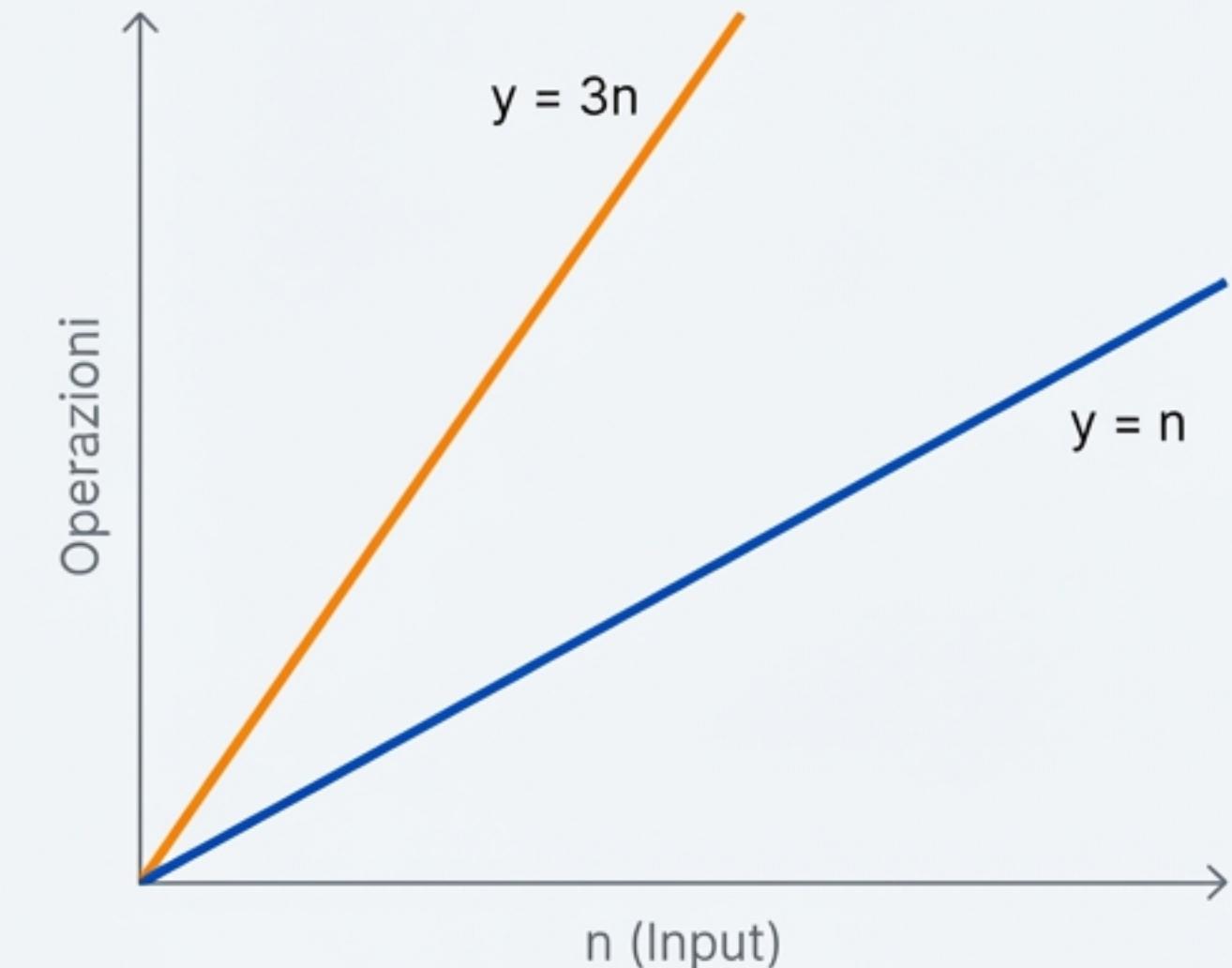
## Regola 1: Si ignorano le costanti.

Non ci interessa il numero esatto di operazioni. Un algoritmo che richiede  $5n$  operazioni e uno che ne richiede  $n$  appartengono alla stessa classe di complessità. La loro crescita è comunque lineare.

### Esempi Pratici:

$$O(2n) \rightarrow O(n)$$

$$O(500) \rightarrow O(1)$$



Entrambe le linee mostrano una crescita lineare. Appartengono alla stessa "famiglia".

## **Le Regole del Gioco (2/2): Concentrarsi sul Termine Dominante**

**Regola 2: Si considera solo il termine più "potente".**

Quando `n` diventa molto grande, l'impatto dei termini a crescita più lenta diventa trascurabile. Cerchiamo il fattore che guida la crescita.

Esempi d'impatto:

$O(n^2 + 500n)$  diventa  **$O(n^2)$**

$O(2^n + n^2)$  diventa  **$O(2^n)$**

$$n^2 + \cancel{500n} \rightarrow O(n^2)$$

# O(1) – Tempo Costante: L'efficienza istantanea

L'operazione richiede sempre lo stesso tempo, indipendentemente dalla dimensione dell'input n.

**Analogia:** Come prendere il primo libro da uno scaffale.  
Non importa se ci sono 10 o 1000 libri, l'azione è la stessa.

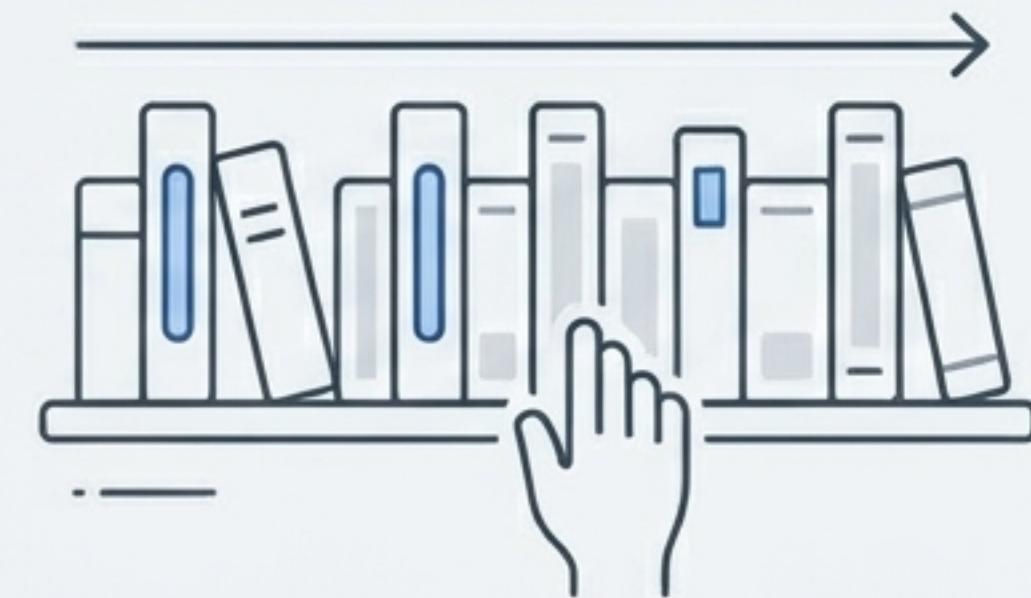


```
// L'accesso a un elemento di un array tramite indice è O(1)  
→ int primoElemento = arr[0];
```

# **O(n) – Tempo Lineare: Crescita prevedibile**

Il tempo di esecuzione (o il numero di operazioni) cresce in modo direttamente proporzionale alla dimensione dell'input.

**Analogia:** Come leggere tutti i titoli dei libri su uno scaffale. Più libri ci sono, più tempo ci vuole.



```
// Un singolo ciclo che itera su tutti gli n elementi  
for (int num : arr) {  
    print(num);  
}
```

# $O(n^2)$ – Tempo Quadratico: Attenzione ai cicli annidati

Il tempo di esecuzione cresce con il quadrato della dimensione dell'input. Diventa rapidamente molto lento per n di grandi dimensioni.

Analogia: Come confrontare ogni libro sullo scaffale con ogni altro libro.



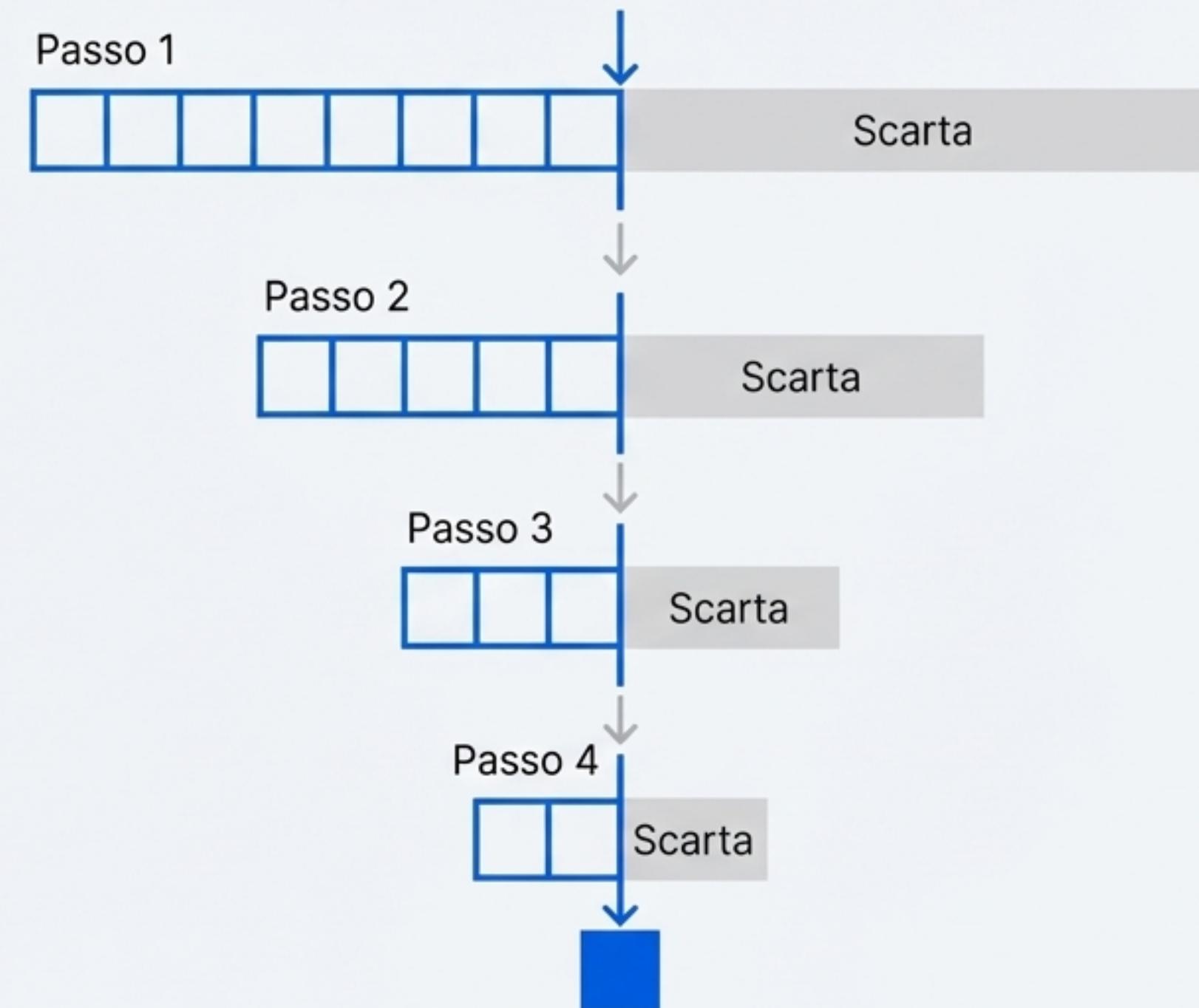
```
// Un ciclo annidato nell'altro, entrambi dipendenti da n
for (int num1 : arr) {                  // <- n volte
    for (int num2 : arr) {              // <- n volte
        n | n                         // ...esegui un'operazione
        }
    } // n * n = n2 operazioni
```

# $O(\log n)$ - Tempo Logaritmico: Il potere della divisione

Estremamente efficiente. Ad ogni passo, l'algoritmo scarta una porzione significativa del problema (spesso la metà).

Analogia: Come cercare una parola in un dizionario. Non leggi ogni pagina; apri a metà e decidi in quale metà continuare la ricerca, dimezzando il problema ad ogni passo.

Esempio Classico: Ricerca Binaria (Binary Search)



# Non solo Tempo: Analizzare l'Uso della Memoria

La Complessità Spaziale misura la memoria *\*aggiuntiva\** allocata dall'algoritmo. Non si conta la memoria occupata dall'input stesso.

## Spazio O(1) (Costante)

Allocata solo un numero fisso di variabili.

```
→ int sum = 0; // Solo una variabile, indipendentemente da n
    for (int num : arr) {
        sum += num;
    }
```

Allocazione di una singola  
variabile (costante)

## Spazio O(n) (Lineare)

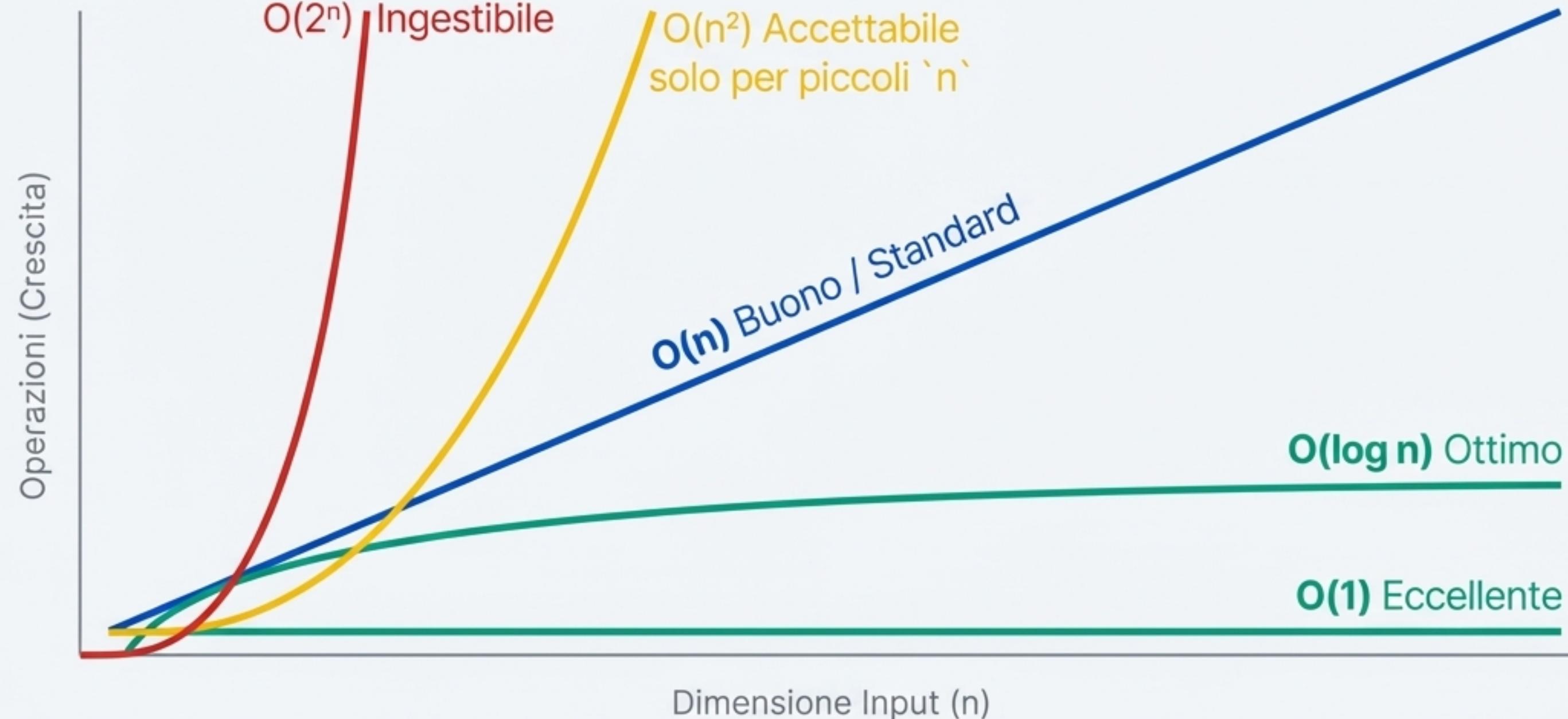
Allocata una quantità di memoria  
proporzionale all'input.

```
// Crea un nuovo array della stessa dimensione dell'input
int[] copiaArray = new int[arr.length];
```



Allocazione di memoria  
proporzionale a 'n' (lineare)

# La Gerarchia delle Performance: Una Guida Visiva



Usa questo **grafico** come guida mentale per valutare rapidamente l'efficienza potenziale dei tuoi algoritmi.



# Da Sviluppatore a Ingegnere del Software

Comprendere la Big O non è un esercizio accademico. È lo strumento che ti permette di prendere **decisioni di design consapevoli**, scrivendo codice che non solo funziona, ma funziona in modo **efficiente e scalabile**.

La prossima volta che scrivi una funzione, fermati un istante e chiediti:

- ✓ Qual è la sua complessità Big O?
- ✓ Come si comporterà con un milione di elementi?
- ✓ Esiste un approccio più efficiente?