

# **COMPLESSITA' COMPUTAZIONALE**

Esercitazioni  
Tutor: Francesca Piersigilli

La *complessità computazionale* si occupa della valutazione del **costo** degli algoritmi in termini di risorse di calcolo:

- **tempo** di elaborazione;
- quantità di memoria (**spazio**) utilizzata.

L'obiettivo è quello di comprendere le prestazioni massime raggiungibili da un algoritmo applicato ad un determinato problema.

# ***COMPLESSITA' SPAZIALE***

Lo spazio utilizzato da un programma può essere visto come la somma di due differenti componenti.

*Spazio fisso:*

- codice
- variabili
- array di dimensioni fisse
- costanti con nome

*Spazio variabile:*

- variabili allocate dinamicamente
- stack

# ***COMPLESSITA' TEMPORALE***

La complessità temporale può essere vista come somma di due componenti:

- tempo di compilazione
- tempo di esecuzione

L'unica quantità che interessa realmente è il tempo di esecuzione:

- richiede una profonda conoscenza del compilatore;
- è strettamente dipendente dalle ottimizzazioni effettuate;
- viene influenzato dall'architettura su cui il programma viene eseguito.

# PASSO DI UN PROGRAMMA

Una stima della complessità temporale di un algoritmo può essere effettuata contando il numero di operazioni svolte.

Si definisce ***PASSO DI UN PROGRAMMA*** (***Program Step***) un segmento di codice autonomo e coerente dal punto di vista semantico o sintattico.

La valutazione delle prestazioni mediante il conteggio dei passi di programma eseguiti, è indipendente sia dal calcolatore utilizzato che dal compilatore.

# PRECISIONE

- Come si può intuire, l'utilizzo dei passi di programma per l'analisi della complessità non consente una valutazione precisa.
- D'altro canto, la determinazione esatta della complessità non è sempre necessaria o possibile.

**Calcoleremo la complessità di un algoritmo considerando solamente il fattore tempo.**

# COMPLESSITA'

Dato un algoritmo  $A$  la complessità viene determinata *contando il numero di operazioni* aritmetiche e logiche, accesso ai file, letture e scritture in memoria...

Occorre tener presente che il tempo  $t$  dipende dai seguenti fattori:

- la *macchina usata*, intendendo con ciò l'hardware, il sistema operativo, il compilatore e, in macchine multitasking, il carico;
- la *configurazione dei dati in ingresso*;
- la *dimensione dei dati in ingresso*.

## **Ipotesi semplificativa:**

Il tempo impiegato è proporzionale al numero di operazioni eseguite (ciascuna a costo unitario), non ci si riferisce, quindi, ad una specifica macchina, ma ci si basa su un modello astratto di calcolatore.

Occorre svincolarsi da una particolare configurazione dei dati in ingresso, ad esempio basandosi sulla configurazione che rappresenta il caso peggiore, ovvero quello che comporta il maggior costo computazionale.

Il tempo impiegato per risolvere un problema dipende sia dall'algoritmo utilizzato che dalla "*dimensione*" *dei dati* a cui si applica l'algoritmo. Quindi nell'ipotesi semplificativa, il tempo deve essere una funzione della dimensione dell'input.



Valutare la complessità degli algoritmi ci consente di scegliere tra loro quello più efficiente (a minor complessità).

La *complessità* dell'algoritmo viene espressa in funzione della *dimensione delle strutture dati*.

ESEMPIO: algoritmo del generico problema P

$$timeAlg(P)(n)=2n$$

è molto diverso da

$$timeAlg(P)(n)=4n^3$$

perché **cambia l'ordine di grandezza** del problema.

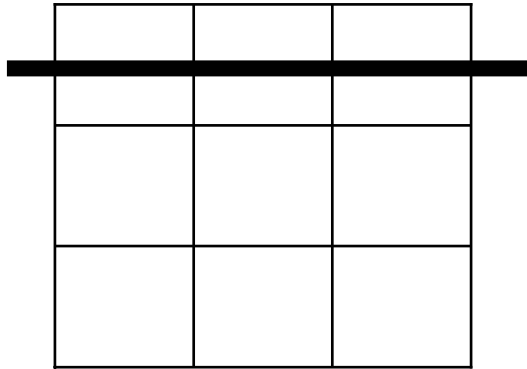
# ESEMPIO

Moltiplicazione di due matrici quadrate  $n \times n$  di interi:  
per calcolare  $C[i,j]$  si eseguono  $2n$  letture,  $n$  moltiplicazioni,  $n-1$  addizioni  
ed 1 scrittura

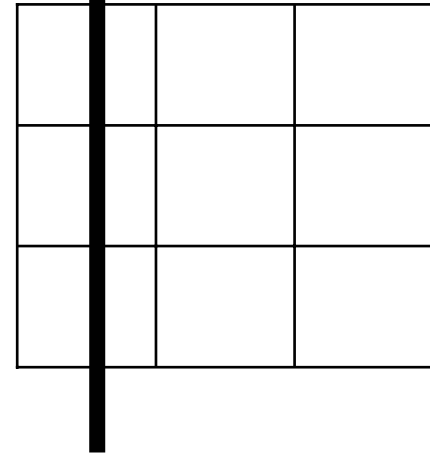
$$n = 3$$

$$C = A \times B$$

A




B




$$c[1,1] = a[1,1]*b[1,1] + a[1,2]*b[2,1] + a[1,3]*b[3,1]$$

Per calcolare un generico  $c[i,j]$  si eseguono, quindi  $4n^3$ :  
 $2n^3 + n^3 + n^2(n-1) + n^2$

# Comportamento asintotico: notazione $O$

Individuare con esattezza la funzione  $time(n)$  è spesso molto difficile.

Spesso però, è sufficiente stabilire il comportamento asintotico della funzione quando le dimensioni dell'ingresso tendono ad infinito (**comportamento asintotico dell'algoritmo**).

# Definizione

Un algoritmo (programma) ha costo  $O(f(n))$  (o complessità  $O(f(n))$ ) se esistono opportune costanti  $a, b, n'$  tali che:

$$timeA(n) \leq a * f(n) + b \quad \text{per ogni } n > n'$$

$O(f(n))$  si dice *limite superiore al comportamento asintotico* di una funzione.

## Esempi:

$time(n) = 3n^2 + 4n + 3 = O(n^2)$ , perché  $3n^2 + 4n + 3 \leq 4n^2$ ,  
per  $n > 3$

$time(n) = 4n^3 = O(n^3)$

Attraverso la notazione  $O()$ , gli algoritmi vengono divisi in **classi**, ponendo nella medesima classe quelli la cui *complessità è dello stesso ordine di grandezza*.

# ***Classi di complessità (I)***

## **1) Complessità *costante***

**$O(1)$**

È posseduta dagli algoritmi che eseguono sempre lo stesso numero di operazioni indipendentemente dalla dimensione dei dati.

Es: inserimento o estrazione di un elemento dalla testa di una lista concatenata, ecc.

## **2) Complessità *sottolineare***

**$O(kn)$ ,  $k < 1$**

Es: Ricerca binaria, o logaritmica, ecc.

# ***Classi di complessità (II)***

## 3) Complessità ***lineare***

**$O(n)$**

È posseduta dagli algoritmi che eseguono un numero di operazioni sostanzialmente proporzionale ad  $n$ .

Es.: ricerca sequenziale, somma di numeri di  $n$  cifre, ecc.

## 4) Complessità ***$n \log n$***

**$O(n \log n)$**

Es.: Algoritmi di ordinamento ottimi.

# ***Classi di complessità (III)***

5) Complessità  $n^k$

$O(n^k)$ , con  $k \geq 2$

Es: Bubblesort ( $O(n^2)$ ), moltiplicazione di due 2 matrici ( $O(n^3)$ ), ecc.

Tutte le classi di complessità elencate precedentemente vengono genericamente considerate come polinomiali. Esse sono caratterizzate dal fatto che la dimensione  $n$  non compare mai come esponente in alcun modo. Quando ciò avviene si parla invece di complessità esponenziale....



## 6) Complessità *esponenziale*

$O(k^n)$  cioè esponenziale.

Es: Torre di Hanoi  $O(2^n)$

- Possono richiedere tempi di esecuzione grandissimi anche per  $n$  piccoli ed indipendentemente dalla velocità dell'elaboratore.
- Esistono molti problemi per i quali non si conoscono ancora algoritmi non esponenziali (problemi intrattabili), come il problema del commesso viaggiatore.
- Un problema è computazionalmente **trattabile** se esiste un algoritmo **efficiente** che lo risolve; altrimenti il problema è computazionalmente **intrattabile**.
- Un algoritmo si dice **efficiente** se il suo tempo di esecuzione è limitato superiormente da una funzione polinomiale (funzione limitata superiormente da  $n^k$  per un opportuno fissato intero  $k$ ).

## ESEMPI:

**$\log n$**  : complessità logaritmica

**$c_1 * n + c_2$**  : complessità lineare

**$n^2 + n$**  : complessità quadratica

**$n^k + c^3 n$**  : complessità polinomiale

**$2^n + n$**  : complessità esponenziale

# ESEMPIO

Dato un problema  $P$  e due algoritmi  $A1$  e  $A2$  che lo risolvono siamo interessati a determinare quale ha complessità minore (qual è "il migliore" dal punto di vista dell'efficienza computazionale).

$$\text{timeA1}(n) = 3n^2 + n$$

$$\text{timeA2}(n) = n^2 + 10n$$

Per  $n \geq 5$ ,  $\text{timeA2}(n) < \text{timeA1}(n)$ . La complessità di  $A2$  è minore per ingressi con dimensione maggiore di 5.

NB: non è detto che quel che vale per  $n$  "grande" valga anche per  $n$  "piccolo". Ma ci interessa che un algoritmo sia efficiente per  $n$  grande (se  $n$  "piccolo", c'è poca differenza, e comunque ogni algoritmo va bene... o quasi!)

# Comportamento asintotico: notazione $\Omega$

## Definizione

Un algoritmo (programma) ha costo  $\Omega(g(n))$  (o complessità  $\Omega(g(n))$ ) se esiste una opportuna costante positiva  $c$  tale che:

$$timeA(n) > c * g(n)$$

*per un numero infinito di valori di  $n$ .*

## Esempio

$$3n^2 + n = \Omega(n^2) \text{ (ma anche } \Omega(n) \text{ e } \Omega(n * \log n))$$

$\Omega(g(n))$ , rappresenta un limite inferiore al comportamento di una funzione.

**Si ha una valutazione esatta del costo di un algoritmo quando le due delimitazioni  $O(f(n))$  e  $\Omega(f(n))$  coincidono.**

In particolare diremo che un problema ha complessità:

➤ ***lineare*** quando ogni algoritmo che lo risolve ha complessità  $O(n)$  e  $\Omega(n)$ ;

➤ ***polinomiale*** quando ogni algoritmo che lo risolve ha complessità  $O(n^k)$  e  $\Omega(n^k)$ , con  $k > 1$ ;

➤ ***Problema intrattabile***: è un problema risolubile, ma per il quale non esiste alcun algoritmo con complessità polinomiale che lo risolve (ad esempio, problema del commesso viaggiatore).

# ISTRUZIONE DOMINANTE

Permette di semplificare in modo drastico la valutazione della complessità.

## ***Definizione***

Dato un algoritmo (o programma) A il cui costo di esecuzione è  $t(n)$ , un'istruzione di A è detta *dominante* quando, per ogni intero  $n$ , essa viene eseguita (nel caso di input con dimensione  $n$ ) un numero  $d(n)$  di volte tale che:

***$t(n) < a d(n) + b$ , per opportune costanti  $a, b$ .***

In pratica, una istruzione dominante viene eseguita un numero di volte proporzionale al costo dell'algoritmo.

Perciò, se esiste un'istruzione dominante, la complessità dell'algoritmo si riconduce a quella legata all'istruzione dominante, ossia la complessità dell'algoritmo è  $O(d(n))$ .

## Esempio:

Ricerca esaustiva di un elemento in un vettore di dimensione  $N$ .

```
boolean ricerca (vettore vet, int el)
{
    int i=0;
    boolean T=falso;
    while (i<N)    /* (1) */
    {
        if (el==vet[i])    /* (2) */ T=vero;
        i++;
    }
    return T;
}
```

Le istruzioni (1) e (2) sono quelle dominanti, e vengono eseguite rispettivamente  $N+1$  volte (per  $i=0..N$ ) e  $N$  volte (per  $i=0..N-1$ ): dunque il costo è lineare.

## Esempi su vettori:

Ricerca del valore minimo e massimo di un vettore

```
int minimo (vettore vet) {  
    int i, min;  
    for (min = vet[0], i = 1; i < N; i ++)  
        if (vet[i]<min) /* istr. dominante */  
            min = vet[i];  
    return min;  
}
```

```
int massimo (vettore vet) {  
    int i, max;  
    for (max = vet[0], i = 1; i < N; i ++)  
        if (vet[i]>max) /* istr. dominante*/  
            max=vet[i];  
    return max;  
}
```

Sia per la ricerca del minimo che del massimo, l'istruzione dominante viene eseguita  $N-1$  volte. Perciò entrambi gli algoritmi hanno un costo  $O(n)$ , se  $n$  è la dimensione del vettore.



# Dipendenze dai dati d'ingresso

Molto spesso il costo dell'esecuzione di un algoritmo dipende non solo dalla dimensione dell'ingresso, ma anche dai particolari valori dei dati in ingresso.

E' il caso tipico di molti algoritmi di ordinamento: se il vettore è già ordinato, finiscono subito, mentre se è "molto disordinato" devono eseguire molti più passi.

Si distinguono diversi casi: caso migliore, caso peggiore, caso medio. Di solito la complessità viene valutata nel caso peggiore (e talvolta nel caso medio).

***Esempio:*** Per la **ricerca sequenziale** in un vettore il costo dipende dalla posizione dell'elemento cercato.

**Caso migliore:** l'elemento è il primo del vettore (un solo confronto)

**Caso peggiore:** l'elemento è l'ultimo o non è presente: l'istruzione dominante è eseguita  $N$  volte ( $N$  dimensione del vettore). Il costo è **lineare**.

## ***OSSERVAZIONE***

Il caso peggiore è utile, ma non sempre significativo in pratica, perché solitamente è anche raro. Ha quindi senso cercare di capire cosa succede "di solito", in un caso "medio".

***Caso medio:*** ciascun elemento è equiprobabile  
***L'elemento cercato potrebbe essere il primo ( 1 confronto), o il secondo (2 confronti), ..., oppure l'ultimo ( N confronti)***