Complessità

I. Studio della Complessità

Lo **studio della complessità** risulta utile per stabilire quali problemi ammettono una soluzione algoritmica e quali invece no.
Per quei problemi che ammettono soluzione, cioè quelli computabili, è utile conoscere anche la **complessità**.

Usando queste informazioni si possono comparare gli algoritmi e definire quali sono i problemi effettivamente computabili.

1.1 - Complessità, Efficienza e Risorse

La **complessità di un algoritmo** è una misura della quantità di risorse utilizzate durante la computazione.

L'**efficienza di un algoritmo** è inversamente proporzionale alla complessità: un programma è più efficiente di un altro se utilizza meno risorse di calcolo.

Le **risorse di calcolo** che si considerano sono:

- tempo di elaborazione **(complessità in tempo)**;
- quantità di memoria necessaria **(complessità in spazio)**.

1.2 - Valutare l'Efficienza

Il tempo di calcolo **non può essere valutato in secondi** perché bisogna considerare le condizioni nelle quali si svolgono le prove: elaboratore utilizzato, compilatore etc. sono fattori che influenzano pesantemente il risultato.

Attraverso la **teoria della complessità** si studia in modo oggettivo l'uso delle risorse necessarie alla computazione dell'algoritmo, dal momento che la complessità è legata all'algoritmo stesso.

Si sceglie poi l'algoritmo che si comporta meglio al crescere della complessità del problema.

Bisogna quindi definire un **modello di costo** che dipende dal particolare modello di macchina astratta a cui si fa riferimento.
In queste macchine, ad ogni operazione è associato un costo (indipendente dal sistema, visto che la macchina è astratta).

Ci si limita a contare solo alcune operazioni chiave.

II. Analisi della Complessità

2.1 - Analisi della Complessità: Dimensione dell'Input

La **dimensione dell'input** influenza il costo di esecuzione: allora la dimensione dell'input rappresenta l'argomento della funzione che esprime il costo di esecuzione del programma.

Questo intero ***n***, che rappresenta la dimensione dell'input, su una **macchina di Turing**
equivale al numero di celle occupate sul nastro di input, mentre su un **elaboratore moderno** sarà lo spazio occupato nella memoria (o un numero che lo rappresenta).

Infatti, nel caso dell'elaboratore moderno, se operiamo con matrici, ***n*** sarà il numero di elementi della matrice, su un grafo sarà il numero di nodi/archi e così via.

2.2 - Analisi della Complessità: Complessità in Spazio

La **complessità in spazio** è il massimo spazio nella memoria usato durante l'esecuzione (sia da dati iniziali/finali che dati di lavoro).

Dal momento che le abbiamo memorie enormi e poco costose, ci si limita allo studio della complessità in tempo.

Per i **dati semplici** il costo sarà unario, mentre per array, record etc. avremo che il costo sarà ***n***
per ogni elemento in essi *(es. array di 5 elementi = 5n)*.

2.3 - Analisi della Complessità: Complessità in Tempo

Fissata la dimensione ***n***, l'obiettivo sarà ora esprimere la **complessità in tempo** come funzione di ***n***, e spesso ci si limita a studiare il comportamento di questa funzione al crescere di ***n*** (detta anche **complessità asintotica**).

In questo processo si tralasciano, ad esempio, le costanti moltiplicative, in quanto poco rilevanti.

2.4 - Analisi della Complessità: Modello di Costo

Come **modello di costo** si usa quello di un linguaggio di programmazione lineare.

Le **operazioni** che hanno **costo unitario** sono:

- assegnazione
- aritmetica (+, -, *, /)
- logica (AND, NOT, OR)
- confronto (>, <, =, etc.)
- lettura/scrittura

Il costo di **operazioni composte** sarà pari alla somma delle operazioni che le compongono.

2.5 - Esempi di Calcolo della Complessità Temporale “O Grande”

Espressione

- $T(n) = 2n^2 + 3n + 1$

Rimuovere i termini di ordine inferiore: $T(n) = 2n^2 \cancel{+ 3n + 1}$
Rimuovere la costante moltiplicativa: $T(n) = \cancel{2}n^2$

Risultato: $T(n) = O(n^2)$

Loop

- $\text{for(i=1; i} \leq \text{n; i++) } \{$
  $x = y + z$
  $\}$

L’espressione $x = y + z$ impiega tempo costante c.
Essendo ripetuta n volte impiegherà tempo pari $c \cdot n$.

Escludendo la costante additiva, la complessità temporale sarà $T(n) = O(n)$

Nested Loop

- $\text{for(i=1; i} \leq \text{n; i++) } \{$
  $\text{for(j=1; j} \leq \text{n; j++) } \{$
  $x = y + z$
  $\}$
  $\}$

In questo caso vale quanto detto nell’esempio precedente. Ci sono però due loop, quindi avremo $T(n) = O(n^2)$

Espressioni in Sequenza

1. $a = a + b$ → impiega tempo costante c_1

1. $\text{for(i=1; i} \leq \text{n; i++) } \{$
   $x = y + z$
   $\}$
   → impiega tempo $c_2 \cdot n$

1. $\text{for(j=1; j} \leq \text{n; j++) } \{$
   $c = d + e$
   $\}$
   → impiega tempo $c_3 \cdot n$

Avremo quindi che:

$T(n) = c_1 + c_2\ n + c_3\ n$

$T(n) = n + n = 2n$

$T(n) = O(n)$

III. Configurazioni

Dal momento che fattori come la dimensione dei dati o l’ordine delle istruzioni (dove sono presenti degli if/else) influisce pesantemente sulla complessità, non si usa un’unica funzione, ma bensì tre:

- **Caso Medio**
  Legata alla complessità media.
  Calcolare la **complessità media** è difficile (si considera la complessità dei tre casi e si usa la probabilità).
- **Caso Ottimo**
  Configurazione che dà luogo al minimo tempo di esecuzione.

Semplice da determinare, ma di interesse secondario.

- **Caso Pessimo**
  Configurazione che dà luogo al massimo tempo di esecuzione; sarà un limite superiore per la complessità.

IV. Notazioni

4.1 - $O(n)$

La notazione **O grande**, applicata alla funzione di complessità, delimita superiormente la crescita della funzione e fornisce un indicatore di bontà dell'algoritmo.

4.2 - $Of((n))$

...

V. Istruzione Dominante

Il concetto di istruzione dominante permette spesso di semplificare la valutazione della complessità di un programma.

Un'**istruzione dominante** è un istruzione che viene eseguita un numero di volte proporzionale al costo di esecuzione di tutto l'algoritmo.
Per individuare un'istruzione dominante è sufficiente esaminare le operazioni che sono contenute nei cicli più interni del programma.

VI. Classi di Complessità

In ordine crescente abbiamo:

Costante	$O(1)$
Logaritmica	$O(log(n))$
Lineare	$O(n)$
nlog	$O(n(log(n)))$
Quadratica	$O(n^2)$
Cubica	$O(n^3)$
Esponenziale	$O(a^n) \quad a > 1$

Gli algoritmi con complessità costante sono più efficienti di quelli con complessità logaritmica che a loro volta sono più efficienti di quelli con complessità lineare e così via.