5 - Complessità Computazionale

Teoria della complessità

Lo studio della **computabilità** aiuta a stabilire **quali problemi ammettono una soluzione** algoritmica e quali no.
Per i problemi computabili dobbiamo conoscere **la complessità degli algoritmi che li risolvono**

La **complessità di un algoritmo** è una misura delle risorse di calcolo consumate durante la computazione.
L'efficienza di un algoritmo è inversamente proporzionale alla sua complessità.

La teoria della complessità studia l'uso delle risorse necessarie per la computazione di algoritmi e per la risoluzione dei problemi.
Questo studio è importante perché:

- Permette di confrontare diversi algoritmi che risolvono lo stesso problema.
- Stabilire se il problema permette un algoritmo che lo risolva in maniera computazionale.

Valutazione dell'efficienza di programmi

Un programma è più efficiente di un altro se la sua esecuzione richiede meno risorse di un altro.
Le risorse di un calcolatore sono:

- Il **tempo** di elaborazione richiesto.
- Il quantitativo di **memoria** necessaria.
  Non possiamo utilizzare per esempio i **secondi** come metodo di valutazione o unità di calcolo.
  Il metodo proposto non è attendibile perché bisogna considerare le condizioni in cui si effettuano le prove, per esempio:
- L'elaboratore su cui il programma viene eseguito
- Il compilatore utilizzato per la traduzione.
- La modalità di ingresso dei dati.
- La significatività dei dati sui quali è eseguita la prova.

Utilizzando una valutazione come i secondi non otterremo una **valutazione oggettiva** del costo di esecuzione di quel programma.

Complessità computazionale

La teoria della complessità studia l'uso delle risorse di calcolo necessarie per la computazione di algoritmi.

Per risolvere un programma spesso sono disponibili vari algoritmi, per la scelta dell'algoritmo giusto o si valuta la bontà dell'algoritmo o si **confrontano più algoritmi più algoritmi sulla base del comportamento che questi presentano al crescere della dimensione del problema**.

Le risorse principali di nostro interesso sono **tempo** e **spazio di memoria**.
Per un algoritmo si parla di:

- Complessità in tempo.
- Complessità in spazio.
  Per poter valutare la complessità occorre definire un **modello di costo** che dipende dal particolare modello della macchina astratta su cui si fa riferimento.

L'analisi dell'efficienza di un programma è basata sull'ipotesi che **il costo di ogni istruzione semplice e di ogni operazione di confronto sia pari a 1**, indipendentemente dal linguaggio e dal sistema usato.
Noi conteremo **le operazioni eseguite o alcune operazioni chiave o preminenti** (ammettendo che il tempo complessivo di esecuzione sia direttamente proporzionale al numero di tali operazioni).
Tratteremo come **non significative le costanti moltiplicative** e studieremo le funzioni di complessità nel loro ordine di grandezza.

La dimensione dell'input

Il costo di esecuzione di un programma dipende anche dal tipo **tipo di dato in ingresso**.
Per esempio il costo di esecuzione di un programma di ordinamento di un insieme di numeri dipende dalle **dimensioni** dell'insieme che si considera.
Per tener conto del numero di dati con cui si esegue il programma assumiamo che **dimensione dell'input rappresenti l'argomento della funzione che esprime il costo** di esecuzione di un programma.

In una Macchina di Turing, tale dimensione può essere caratterizzata da un intero n, che identifica la lunghezza della porzione di nastro contenente i dati di ingresso.
Impiegando un elaboratore con un linguaggio di programmazione, la dimensione n è lo spazio occupato nella memoria dell'elaboratore dai dati relativi al problema da risolvere, generalmente un numero proporzionale a questo spazio.
Esempi:

- Se si opera su matrici, n è il numero dei loro elementi.
- Se si opera su un insieme, n è il numero dei suoi elementi.
- Se si opera su un grafo, n è il numero dei nodi o il numero di archi o la somma di entrambi.

Complessità in spazio

La complessità in **spazio** è il massimo spazio occupabile in memoria durante l'esecuzione dell'algoritmo, il quale può costruire insiemi di dati intermedi o di servizi, oltre ad operare sui dati iniziali e finali.
Siccome abbiamo delle memorie grandissime a basso costo, **studieremo la complessità in tempo**; **tuttavia, è importante evitare di presupporre la disponibilità di una memoria infinita**.

Riguardo ai costi:

- Il costo è pari a 1 per i dati di tipo semplice.
- Per un array di n elementi, il costo è dato da n \times \text{dimensioni di un elemento} .
- Il costo di un record è determinato dalla somma dei costi delle sue singole componenti.

Esempio: L'algoritmo

```
somma = 0
for i = 1 to n
somma = somma + A[i]
```

ha una complessità pari a 13, ipotizzando che n=10. (cioè: somma = 1, n =1, i=1, gli n elementi=10)

Complessità in tempo e asintotica

Asintotica significa che n tende a infinito:
Fissata una dimensione n, il tempo che un algoritmo impiega per risolvere il problema definisce la **complessità in tempo**.
L'obbiettivo principale sarà quello di **esprimere la complessità in tempo come funzioni di** n.
Studieremo il comportamento di tale funzione al crescere di n(complessità asintotica o semplicemente complessità), considerando i soli termini prevalenti e tralasciando anche costanti moltiplicative.

Lo studio della **complessità asintotica** è motivato dal fatto che gli algoritmi sono sempre definiti per un n generico.

- Per valori **piccoli** di n, non ha senso confrontare l'efficienza di un algoritmo (prendi uno qualsiasi basta che funzioni).
- Per valori n **grandi**, o superiori ad una certa soglia n_0 sarà sempre quello che ha il termine di **grado massimo più basso** a richiedere minor tempo di esecuzione.

Il modello di costo

Assumiamo che il costo di esecuzione di ogni **istruzione semplice** sia unitario, ovvero pari ad 1, per esempio:

- Assegnazione: $\leftarrow$
- Aritmetica: $+, -, *, /$
- Logica: AND, OR, NOT
- Confronto: $\leq, \geq, <, >, =, \neq$
- Lettura e/o scrittura
  Il costo di esecuzione di una **istruzione composta** è pari alla somma dei costi delle istruzioni che la compongono.
- **Costo di un'operazione di selezione (if-then-else):**
  *if **cond** then $S1$ else $S2$ è dato da:*

```
costo test cond + costo S1 //se cond
costo test cond + costo S2 //se ¬ cond
```

Per una struttura if cond then **S1** else **S2**, il costo dipende dal risultato della condizione:

- Se **cond** è vera, il costo è costo test cond+costo S1.
- Se **cond** è falsa, il costo è costo test cond+costo S2. Il costo del test della condizione è 1 (dato che è un confronto), e i costi di S1 e S2 dipendono dalle istruzioni che contengono.
- **Costo di un'istruzione di ciclo (while):**
  *while **cond** do $S1$ è dato da:*

```
costo test cond + (costo test cond + costo S1) * k //[se il ciclo è
ripetuto k volte]
```

- **Costo di un ciclo "repeat":**
  *repeat **S1** until **cond** è dato da:*

```
  (costo test cond + costo S1) * k //[se il ciclo è ripetuto k volte]
```

Per una struttura repeat S1 until cond, il costo è dato da:

- (costo test cond+costo S1)×k dove k è il numero di volte che il ciclo viene ripetuto. In questo caso, il test della condizione avviene alla fine di ogni iterazione, il costo totale è la somma dei costi di S1 e del test, ripetuti k volte.
- **Costo di una istruzione for:**

```
i = 1  (costo = 1, per l'assegnazione iniziale)
while i <= k do
     S1;  (costo dipende dal contenuto di S1)
     i = i + 1;  (costo = 2, per l'assegnazione e l'incremento)
```

Il costo iniziale fisso è 2: 1 per $i = 1$ e 1 per la prima valutazione della condizione while fuori dal ciclo (che avviene prima di entrare nel ciclo).
Per ogni iterazione del ciclo, il costo è dato dal costo di $S1 + 2$(da i=i+1)$+1$
(da condizione di test). Moltiplicando questo costo per il numero di iterazioni k, otteniamo la parte variabile.
Quindi, il costo totale è: $2 + (costo\ di\ S1 + 2 + 1) \times k$ che si semplifica in: 2+(costo\ di \ S1+3)×k

Complessità e configurazioni

La complessità di un algoritmo non può sempre essere definita da una singola funzione, poiché il tempo di esecuzione, a parità di dimensione dei dati, può variare in base alla specifica configurazione dei dati.
Di solito, si analizzano tre tipologie di complessità:

- **CASO PESSIMO:** consideriamo la configurazione che da luogo al massimo tempo di calcolo per una data dimensione n, questa funzione di complessità si indica con: $f_{pess}(n)$.
- **CASO OTTIMO:** identifica la configurazione che minimizza il tempo di calcolo, con funzione di complessità $f_{ott}(n)$.
- **CASO MEDIO:** calcola il tempo medio di esecuzione su tutte le possibili configurazioni dei dati per una dimensione n, con funzione di complessità $f_{med}(n)$.

Alcuni algoritmi presentano comportamenti uniformi indipendentemente dalla configurazione dei dati.
Esempi:
$f_{pess} = f_{ott} = f_{med}$ per l'algoritmo fattoriale
$f_{pess} = f_{ott} = f_{med}$ per l'algoritmo del Min di n elementi

Ricerca lineare

La ricerca lineare in un array è un metodo semplice per trovare un elemento specifico k in un array non ordinato.
L'algoritmo scorre tutto l'array dall'inizio alla fine, confrontando ogni elemento fino a trovare una corrispondenza o fino ad esaurire gli elementi.

```
{1}     i ← 0
{2}     repeat
{3}         i ← i + 1
{4}     until ( t [i].chiave = k) or (i = n)
{5}     if ( t [i].chiave = k) then
{6}       trovato ← true
{7}     else trovato ← false
```

Parametri di input e output:

- t: Array di riferimento (contiene gli elementi da cercare, ognuno con un campo chiave)
- k: valore della chiave da cercare (tipicamente un intero o una stringa)
- *trovato*: variabile booleana per riferimento (output: `true` se l'elemento è trovato, altrimenti `false`)
  Descrizione: la procedura cerca l'elemento con chiave k. Se la ricerca ha successo, imposta `trovato` a `true`, altrimenti a `false`. È un approccio che non richiede l'array ordinato, ma può essere inefficiente per array grandi perché nel peggiore dei casi controlla tutti gli elementi.

Il **costo di esecuzione di un algoritmo** dipende dalla **posizione del particolare elemento** che si vuole individuare, se l'elemento cercato è il primo, allora si effettua solo un confronto (**CASO OTTIMO**), se l'elemento cercato è il secondo allora si effettuano due confronti e così via.
Il **CASO PEGGIORE** è costituito dalla ricerca dell'ultimo elemento o da una ricerca infruttuosa, perché in questo caso l'algoritmo esamina tutte le componenti dell'array ed esegue il ciclo n volte.

VALUTAZIONE DETTAGLIATA DEL COSTO PER LA RICERCA DELL'ULTIMO ELEMENTO
Assumiamo che l'elemento cercato sia l'ultimo (posizione n), quindi ricerca fruttuosa ma costosa.

- L'istruzione {1} (i ← 0): Eseguita 1 volta (costo 1).
- L'istruzione {3} (i ← i + 1): Eseguita n volte (costo 2 per volta, Il totale non è il doppio delle n volte, dato che il costo è 2 per volta).
- Il test dell'istruzione {4} **repeat**: Eseguito n volte (ogni iterazione controlla la condizione alla fine del repeat-until, costo 1 per volta).
- Il test dell'istruzione **if-then-else**: Eseguito 1 volta (costo 1).
- L'istruzione {6} (trovato ← `true`): Eseguita 1 volta (costo 1, poiché ultimo elemento trovato).

- L'istruzione {7}: Non viene eseguita.

Costo totale: $1(iniz.) + n(incrementi) + n(test\ until) + 1(if) + 1(assegn.\ true) =$ **3 + 2n**.

- Nota: Il repeat-until esegue il corpo almeno una volta, e il test alla fine. Per n elementi, il ciclo si ripete n volte prima di uscire.
  E' facile vedere che la stessa funzione esprime anche il costo del programma nel caso di ricerca infruttuosa.

Se si vuole valutare il comportamento del programma nel **CASO MEDIO** è necessario distinguere il caso di ricerca con successo da quello di ricerca infruttuosa.
Nel caso di ricerca fruttuosa, **se si assume che tutti gli elementi dell'array possano essere ricercati con uguale probabilità pari a 1/n**, allora il programma richiede mediamente $(n+1)/2$ confronti.
Infatti se ricerchiamo l'i-esimo elemento si effettuano i confronti e quindi il numero di confronti medio è dato dalla formula probabilistica:

$$\sum_{1 \leq i \leq n} Prob(E(i))i = \sum_{1 \leq i \leq n} = \frac{n+1}{2}$$

Dove:

- $Prob(E(i))$: Probabilità che l'elemento sia in posizione *i* (uguale a 1/n).
- i: Numero di confronti necessari per trovare in posizione *i*.

Focus

La **complessità** di un algoritmo è funzione della **dimensione dei dati**, ovvero della mole dei dati del problema da risolvere, l'individuazione della dimensione dei dati è per lo più immediata.
Determinare la complessità in **tempo** (o in **spazio**) di un algoritmo significa determinare una **funzione di complessità** $f(n)$ che fornisca la misura del tempo (o dello spazio di memoria occupato), al variare della dimensione dei dati n. Pertanto, **la dimensione dei dati non è mai da considerare nell'individuazione della complessità ottima, media e pessima**.
Le funzioni di complessità sono caratterizzate da due proprietà:

- Assumono solo **valori positivi**.
- Sono **crescenti** rispetto alla dimensione dei dati.

Comportamento asintotico

Nel calcolo della complessità di un algoritmo non è necessario determinare con precisione la funzione che ne esprime il costo ma è sufficiente analizzarne il **comportamento asintotico**, ossia come varia quando le dimensioni dell'input tendono all'infinito.
In questa analisi si **trascurano le costanti moltiplicative e i termini additivi di ordine inferiore**, poiché hanno un impatto trascurabile al crescere di n.

Due funzioni come $(3+n)$ e $(100n+3027)$ sono considerate equivalenti dal punto di vista asintotico, in quanto crescono entrambe proporzionalmente a n. Tuttavia, sebbene $(n+3)$ rappresenti un costo inferiore rispetto a $(100n+3027)$, ignorare completamente la costante moltiplicativa può risultare una semplificazione eccessiva in certi contesti.

L'eliminazione di costanti e termini minori semplifica notevolmente l'analisi, rendendo più agevole la valutazione e il confronto tra algoritmi, mantenendo comunque risultati affidabili per valori validi per n che tende ad infinito. Lo **studio asintotico** fornisce dunque un criterio oggettivo e coerente per confrontare l'efficienza di due algoritmi diversi.

Esempio $A_1 : f_1(n) = 3n^2 - 4n + 2 \quad \text{e} \quad A_2 : f_2(n) = 2n + 3$

A_1: $f_1(n) = 3n^2 - 4n + 2$ A_2: $f_2(n) = 2n + 3$

$n = 1, 2$ $f_1(n) < f_2(n)$

$n = 3, 4$ $f_1(n) > f_2(n)$

Qual è l'algoritmo più efficiente?

Per piccoli valori di n (ad esempio $n = 1, 2$), si osserva che $f_1(n) < f_2(n)$, quindi in questa fase l'algoritmo A_1 risulta più efficiente.
Tuttavia, al crescere di n (per $n = 3, 4$ e oltre), la funzione $f_1(n)$ supera $f_2(n)$; ciò significa che, per input di dimensione maggiore, A_2 diventa l'algoritmo più efficiente poiché ha un costo di crescita inferiore. In generale, per $n > 2$, vale infatti $f_1(n) > f_2(n)$.

Questo esempio mostra che l'efficienza di un algoritmo può dipendere dal valore di n e che l'analisi asintotica si concentra sul comportamento per n molto grandi, trascurando l'andamento nei casi piccoli.

Le funzioni di complessità $f_1(n) = 3n^2 + n + 1$, $f_2(n) = 4n^2$ e $f_3(n) = 4n^2 + 2n$ sono considerate **asintoticamente equivalenti**, perché tutte crescono proporzionalmente al quadrato di n.
In altre parole, le differenze dovute a costanti moltiplicative o a termini di grado inferiore diventano irrilevanti quando n tende all'infinito.

Lo studio asintotico, infatti, **trascura costanti e termini meno significativi**, concentrandosi solo sul termine dominante, che determina il comportamento di crescita della funzione e quindi la classe di complessità dell'algoritmo.

Valutazione complessità

Regole per trovare la delimitazione superiore della complessità:

1. **REGOLA 1**: Supponiamo che il programma sia composto da due parti P e Q da eseguire **sequenzialmente** e che i costi di P e Q siano:

   $$S(n) = O(f(n)) e T(n) = O(g(n))$$

   Il costo del programma è $O(max(f(n), g(n)))$

2. **REGOLA 2**: La seconda regola ci permette di valutare il costo del programma quando esso richiede **più volte l'esecuzione di un insieme di istruzioni** o l'attivazione di una procedura.
   Supponiamo che un programma richieda k volte l'esecuzione di una istruzione composta o l'attivazione di una procedura, e sia $f_i(n)$ il costo relativo all'esecuzione i-esima, i=1,2...k. Il costo complessivo del programma è pari a:

$$O\left(\sum_i f_i(n)\right)$$

   L'**istruzione dominante** ci permette di semplificare in modo drastico la valutazione della complessità di un programma.
   Def: Sia dato un programma o un algoritmo P il cui costo di esecuzione è $t(n)$.
   Una istruzione di P si dice **istruzione o operazione dominante** se, per ogni intero n, essa viene eseguita, nel caso peggiore di input avente dimensione n, un numero di volte $d(n)$ che verifica la condizione $t(n) < a * d(n) + b$ per opportune costanti a e b.
   Una **istruzione dominate viene eseguita un numero di volte proporzionale al costo di esecuzione di tutto l'algoritmo**.
   È importante ricordare che in un programma più istruzioni possono essere dominanti ma può accadere che il programma non contenga istruzioni dominanti.
3. **REGOLA 3**: Supponiamo che un programma abbia una istruzione dominate che nel caso peggiore di input di dimensione n, viene eseguita $d(n)$ volte; La delimitazione superiore alla complessità del programma è $O(d(n))$ (in molti casi per individuare una istruzione dominante è sufficiente esaminare le **operazioni che sono contenute in cicli interni** del programma).
   Ad esempio, nel caso della ricerca sequenziale considerata in un esempio precedente, è possibile valutare il costo del programma mostrato osservando che l'istruzione {3} è dominante. Infatti essa viene eseguita, nel caso peggiore, n volte; questo è sufficiente per dire che il programma ha costo lineare. Osserviamo inoltre che il test {4} del ciclo repeat until viene eseguito nel caso peggiore n volte e, quindi, rappresenta un'altra istruzione dominante.

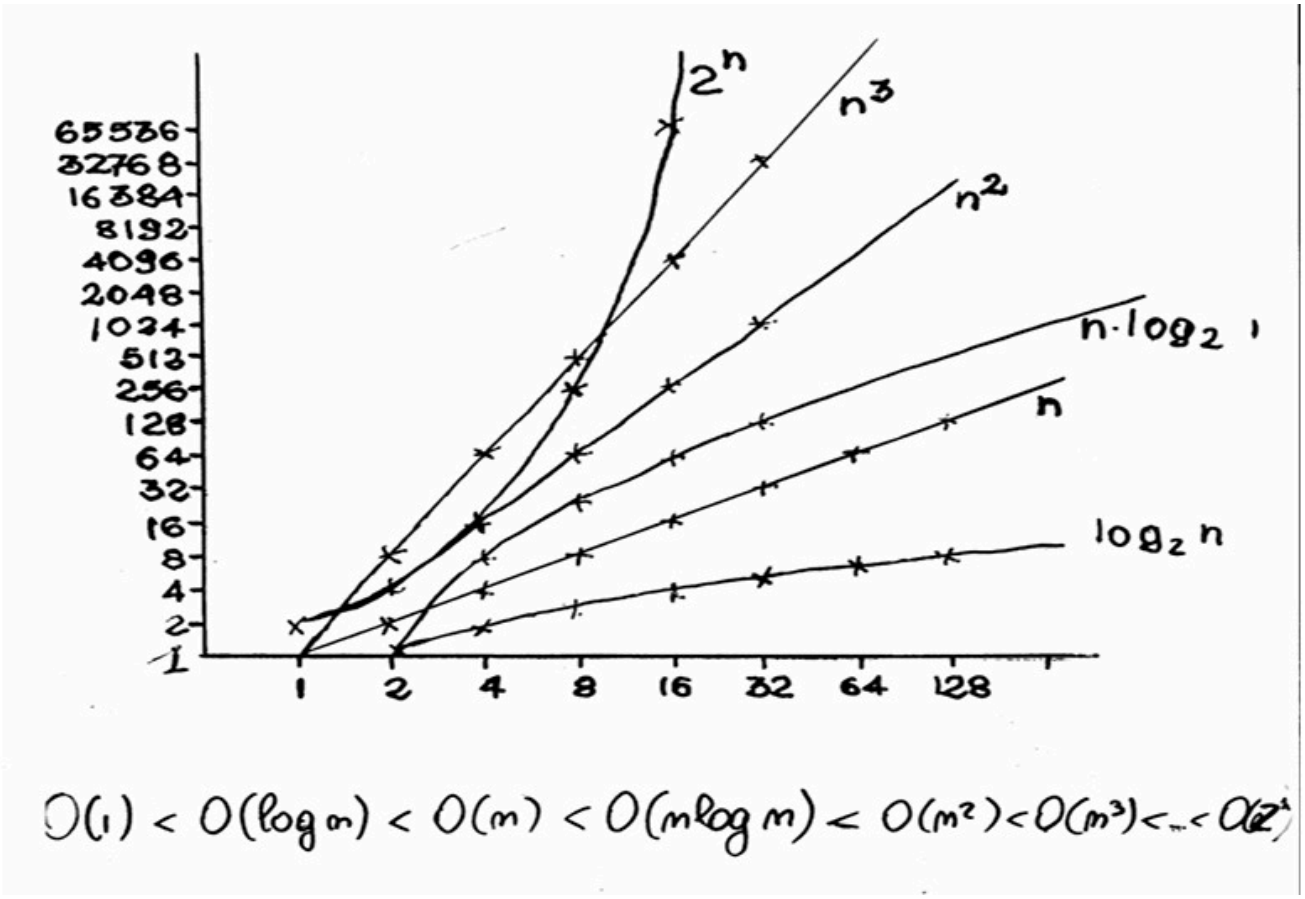
Classi di complessità

Attraverso lo studio della complessità asintotica è possibile suddividere gli algoritmi in diverse **classi di complessità**, ciascuna rappresentativa della loro crescita in funzione della dimensione dell'input.

Nome della classe	Funzione
Costante	$O(1)$
Logaritmica	$O(\log n)$
Lineare	$O(n)$
n log n (Loglineare)	$O(n \log n)$

Nome della classe	Funzione
Quadratica	$O(n^2)$
Cubica	$O(n^3)$
Esponenziale	$O(a^n)$, con $a > 1$

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.
$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(a^n)$



Algoritmi polinomiali vs. esponenziali

La distinzione di maggiore interesse in teoria della complessità è fra:

- **algoritmi polinomiali** complessità $O(n^k), k > 0$
  Esempi sono: $O(log\, n), O(n^{10})$
- **algoritmi esponenziali** complessità $O(a^{g(n)}), a > 1$ e $g(n)$ funzione crescente
  Esempi sono: $O(2^n), O(2^{log\, n}), O(n^{log\, n})$

Gli algoritmi di importanza pratica sono quelli polinomiali