Lezione di Informatica Teorica: Complessità Strutturale

Appunti da Trascrizione Automatica

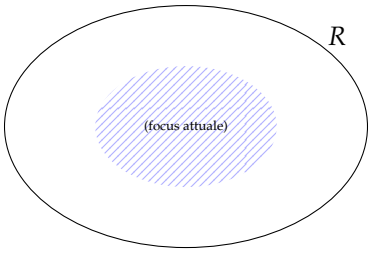
30 giugno 2025

Indice

1 Introduzione alla Complessità Strutturale

Finora, il corso si è occupato principalmente di decidibilità, ovvero di stabilire se un dato problema (linguaggio) è risolvibile o meno (se appartiene a R, RE, $coRE$, o nessuno dei due). L'ultima lezione ha introdotto problemi indecidibili, alcuni più "difficili" di altri (es. il complemento di $HALT$ è in $coRE$, ma $HALT$ non è in R). Tali problemi rientrano nel campo della teoria della ricorsione e della logica matematica.

Da questo momento in poi, l'attenzione si sposterà all'interno della classe R (problemi ricorsivi o decidibili).



L'obiettivo è categorizzare i problemi decidibili in base alla loro *complessità computazionale*, ovvero quanto tempo e/o spazio è richiesto per risolverli. Questo studio prende il nome di **Complessità Strutturale**.

Storicamente, lo studio della decidibilità ha preceduto quello della complessità. Negli anni '30, con i lavori di Turing, l'interesse era capire cosa le macchine astratte (come le Macchine di Turing) potessero risolvere. Con l'avvento dei primi computer reali negli anni '40 (ENIAC, EDVAC), divenne evidente che alcuni problemi erano più veloci da risolvere di altri, spingendo la necessità di una teoria formale per quantificare questa differenza. I lavori seminali sulla complessità computazionale delle Macchine di Turing furono pubblicati nel 1965 da Hartmanis, Stearns e Lewis.

2 Nozioni di Complessità Temporale

Definiamo formalmente il tempo di esecuzione per le Macchine di Turing.

Definizione 2.1 (Computation Time)**.** *Sia M una Macchina di Turing e w una stringa in input per M.*

- *Se M è **deterministica**: il **computation time** di M su w è il numero di passi che M esegue prima di arrestarsi su w.*
- *Se M è **non-deterministica**: il **computation time** di M su w è la lunghezza del* branch di computazione più lungo *(il cammino più lungo nell'albero di computazione di M su w).*

Definizione 2.2 (Time Function)**.** *Una funzione $t : \mathbb{N} \to \mathbb{N}$ è detta **time function** se è non-decrescente e strettamente positiva.*

- ***Non-decrescente**: Se l'input è più grande, il tempo richiesto non diminuisce.*
- ***Strettamente positiva**: Il tempo richiesto è sempre maggiore di zero.*

Definizione 2.3 (Running Time di una Macchina di Turing)**.** *Sia $t(n)$ una time function. Una Macchina di Turing M ha **running time** $t(n)$ se, per tutte le stringhe w in input (a parte un numero finito di esse), il computation time di M su w non eccede $t(|w|)$ (dove $|w|$ è la lunghezza di w).*

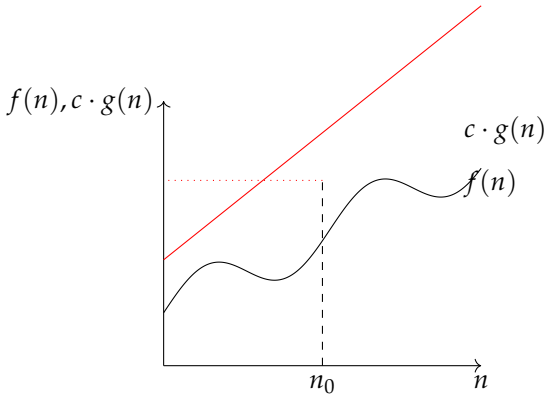
In altre parole, il running time è una stima funzionale del tempo di esecuzione di una macchina, in funzione della dimensione dell'input. Questo concetto è simile alla complessità degli algoritmi che avete studiato.

2.1 Notazione Asintotica

Per esprimere i running time in modo indipendente dalle costanti e concentrarsi sul tasso di crescita, usiamo la notazione asintotica.

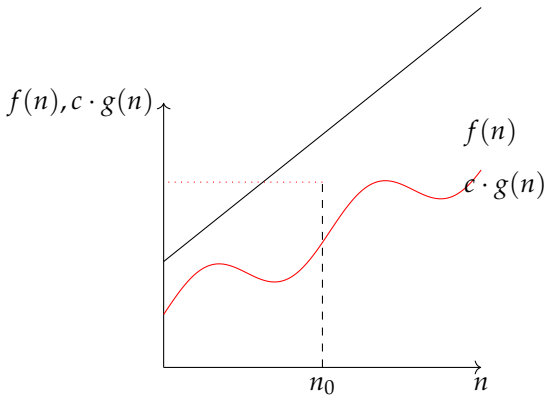
Definizione 2.4 (Big O (O))**.** *Date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$, diciamo che $f(n) \in O(g(n))$ (o $f(n)$ è $O(g(n))$) se esistono costanti positive c e n_0 tali che per ogni $n \geq n_0$, si ha $f(n) \leq c \cdot g(n)$.*

Intuizione: $f(n)$ cresce al più velocemente quanto $g(n)$ (cioè $g(n)$ è un *upper bound* asintotico per $f(n)$).



Definizione 2.5 (Big Omega (Ω))**.** *Date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$, diciamo che $f(n) \in \Omega(g(n))$ (o $f(n)$ è $\Omega(g(n))$) se esistono costanti positive c e n_0 tali che per ogni $n \geq n_0$, si ha $f(n) \geq c \cdot g(n)$.*

Intuizione: $f(n)$ cresce almeno velocemente quanto $g(n)$ (cioè $g(n)$ è un *lower bound* asintotico per $f(n)$).



Definizione 2.6 (Big Theta (Θ))**.** *Date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$, diciamo che $f(n) \in \Theta(g(n))$ (o $f(n)$ è $\Theta(g(n))$) se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$.*

Intuizione: $f(n)$ e $g(n)$ hanno lo stesso tasso di crescita asintotico.

Teorema 2.1 (Linear Speedup (menzionato))**.** *Per ogni Macchina di Turing M che opera in tempo $t(n)$, e per ogni costante $c > 0$, esiste un'altra Macchina di Turing M' che opera in tempo $t(n)/c$. Questo teorema implica che le costanti moltiplicative nel running time non sono significative quando si parla di classi di complessità, poiché si possono sempre "compattare" più simboli sul nastro per accelerare la computazione.*

2.2 Complessità Temporale dei Problemi

I concetti di Big O e Big Omega ci permettono di definire la complessità temporale dei problemi stessi, non solo degli algoritmi.

Definizione 2.7 (Time Complexity Upper Bound di un Problema)**.** *Il **time complexity upper bound** di un problema P è $O(f(n))$ se* esiste almeno un algoritmo *che risolve P con un running time $O(f(n))$.*

Esempio 2.1. *Il problema dell'ordinamento di array:*

- *È $O(n^2)$? Sì, (es. Bubble Sort).*
- *È $O(2^n)$? Sì, (es. Bubble Sort è anche $O(2^n)$, è un upper bound molto lasco ma valido).*
- *È $O(n \log n)$? Sì, (es. Merge Sort).*
- *È $O(n)$? No, si può dimostrare che non esiste un algoritmo di ordinamento basato su confronti che sia $O(n)$.*

Definizione 2.8 (Time Complexity Lower Bound di un Problema)**.** *Il **time complexity lower bound** di un problema P è $\Omega(f(n))$ se* tutti gli algoritmi *che risolvono P hanno un running time $\Omega(f(n))$.*

Esempio 2.2. *Il problema dell'ordinamento di array:*

- *È $\Omega(n)$? Sì, tutti gli algoritmi devono almeno leggere tutti gli elementi.*
- *È $\Omega(n \log n)$? Sì, è dimostrato che questo è il lower bound per algoritmi basati su confronti.*
- *È $\Omega(n^2)$? No, perché esistono algoritmi (es. Merge Sort) che sono $O(n \log n)$, quindi non tutti gli algoritmi sono $\Omega(n^2)$.*

2.3 Problemi Trattabili vs. Intratttabili

Basandoci sulla complessità temporale, i problemi possono essere classificati in:

- **Problemi Trattabili (o Facili)**: Sono problemi il cui time complexity upper bound è *polinomiale*.
- **Problemi Intratttabili (o Difficili)**: Sono problemi per i quali *non esiste* (o non si è ancora dimostrato che esista) un algoritmo con running time polinomiale. Questi problemi generalmente richiedono tempo esponenziale.

Nota: Un algoritmo $O(n^{1000})$ è teoricamente polinomiale, ma impraticabile per grandi n. Tuttavia, algoritmi esponenziali (es. $O(2^n)$) diventano ingestibili molto più rapidamente con l'aumentare di n rispetto a qualsiasi polinomio.

3 Classi di Complessità Temporale

Il prossimo passo è definire classi formali di problemi basate sulla loro complessità temporale, all'interno di R.

Definizione 3.1 (Classe DTIME($t(n)$))**.** *Sia $t(n)$ una time function. La classe di complessità **DTIME($t(n)$)** è l'insieme di tutti i linguaggi L tali che esiste una Macchina di Turing* deterministica *M che decide L con un running time $O(t(n))$.*

La 'D' in DTIME sta per "Deterministica".

Definizione 3.2 (Classe P (Polynomial Time))**.** *La classe **P** è l'unione di tutte le classi DTIME(n^c) per ogni costante $c \geq 1$:*

$$P = \bigcup_{c \geq 1} DTIME(n^c)$$

Nota Importante: La classe P contiene *esclusivamente problemi di decisione*. Problemi di calcolo (es. sommare due numeri) o di ottimizzazione (es. ordinare un array) non appartengono a P per definizione, sebbene gli algoritmi per risolverli possano avere complessità polinomiale.

3.1 Esempi di Problemi in P

Esempio 3.1 (Reachability (Raggiungibilità))**.**

- ***Problema di Decisione**: Dato un grafo diretto $G = (V, E)$ e due nodi $s, t \in V$, esiste un percorso da s a t in G?*
- ***Linguaggio**: $L_{Reach} = \{(G, s, t) \mid G$ è un grafo diretto, $s, t \in V(G)$, e esiste un percorso da s a t in $G\}$.*
- ***Algoritmo**: È possibile risolvere Reachability usando algoritmi come la Breadth-First Search (BFS) o la Depth-First Search (DFS).*
  - ***Funzionamento intuitivo di BFS/DFS**: Si parte dal nodo s, si visitano tutti i nodi adiacenti, poi i nodi adiacenti dei nodi visitati, e così via, segnando i nodi già visitati per evitare cicli. Se t viene raggiunto, la risposta è "sì".*
  - ***Complessità**: La BFS/DFS ha una complessità temporale di $O(V + E)$ (dove V è il numero di vertici ed E è il numero di archi), che è polinomiale nella dimensione del grafo (input).*
- ***Conclusione**: Poiché esiste un algoritmo deterministico con complessità polinomiale, $L_{Reach} \in P$.*

Esempio 3.2 (PRIMES)**.**

- ***Problema di Decisione**: Dato un intero n (rappresentato in binario), è n un numero primo?*
- ***Linguaggio**: $L_{Primes} = \{\langle n \rangle \mid n \in \mathbb{N}$ e n è primo$\}$, dove $\langle n \rangle$ è la rappresentazione binaria di n.*
- ***Algoritmo Naïf (Test di Divisione)**: Per verificare se n è primo, si può tentare di dividerlo per tutti gli interi da 2 fino a $\sqrt{n}$.*
  - ***Numero di Divisioni**: Circa $\sqrt{n}$.*
  - ***Costo di ogni Divisione**: Per numeri grandi, una divisione non è una operazione a costo costante, ma richiede tempo polinomiale nella lunghezza dei numeri coinvolti.*
  - ***Complessità Totale**: Il numero di divisioni è $\sqrt{n}$. Se l'input è la stringa binaria $\langle n \rangle$, la sua lunghezza è $L = |\langle n \rangle| \approx \log_2 n$. Quindi $n \approx 2^L$. Il numero di divisioni è $\sqrt{n} = \sqrt{2^L} = 2^{L/2}$.*

  - ***Conclusione sul Naïf**: Questo algoritmo è* esponenziale *nella lunghezza dell'input (L), non polinomiale.* Attenzione*: un errore comune è confondere il valore dell'input con la sua lunghezza. Contare fino a n o iterare n volte è esponenziale se n è dato in binario.*

- ***Risultato Moderno**: Nonostante l'algoritmo naïf sia esponenziale, il problema PRIMES è stato dimostrato essere in P nel 2002 (Algoritmo AKS, Agrawal, Kayal, Saxena). La sua complessità è $O(L^k)$ per un k piccolo (es. $O(L^6)$ o $O(L^{12})$), rendendolo un problema polinomiale. Tuttavia, in pratica si preferiscono algoritmi randomizzati più veloci (es. Miller-Rabin).*

4 Complessità Temporale Non-Deterministica

Alcuni problemi, pur sembrando difficili con algoritmi deterministici, beneficiano enormemente del non-determinismo.

Esempio 4.1 (SAT (Boolean Satisfiability))**.**

- ***Formule in Forma Normale Congiuntiva (CNF)**: Una formula CNF è una congiunzione di clausole, dove ogni clausola è una disgiunzione di letterali. Un letterale è una variabile booleana o la sua negazione (es. $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_5 \vee \neg x_6)$).*

- ***Problema di Decisione**: Data una formula ϕ in CNF, è ϕ soddisfacibile (esiste un assegnamento di verità alle variabili che rende ϕ vera)?*

- ***Linguaggio**: $L_{SAT} = \{\phi \mid \phi$ è una formula CNF e ϕ è soddisfacibile$\}$.*

- ***Verifica di un Assegnamento**: Se ci viene dato un assegnamento di verità, verificare se soddisfa la formula richiede tempo polinomiale (lineare nella lunghezza della formula).*

- ***Algoritmo Naïf Deterministico**: Testare tutti i possibili assegnamenti di verità. Se ci sono n variabili, ci sono 2^n assegnamenti. Ognuno richiede tempo polinomiale per la verifica.*

- ***Complessità Naïf**: $O(2^n \cdot poly(|\phi|))$. Questo è esponenziale.*

Esempio 4.2 (Independent Set (Insieme Indipendente))**.**

- ***Definizione**: Dato un grafo non diretto $G = (V, E)$, un* independent set *è un sottoinsieme di vertici $S \subseteq V$ tale che nessun vertice in S è collegato da un arco a un altro vertice in S.*

- ***Problema di Decisione**: Dato un grafo G e un intero k, esiste un independent set in G di dimensione almeno k?*

- ***Linguaggio**: $L_{IS} = \{(G, k) \mid G$ è un grafo e esiste un independent set di taglia $\geq k$ in $G\}$.*

- ***Verifica di un Insieme**: Dato un sottoinsieme di vertici S, verificare se è un independent set di taglia $\geq k$ richiede tempo polinomiale (es. $O(V^2)$ o $O(V + E)$).*

- ***Algoritmo Naïf Deterministico**: Testare tutti i possibili sottoinsiemi di vertici di dimensione $\geq k$. Il numero di tali sottoinsiemi può essere $\binom{V}{k}$, che è esponenziale in V.*

- ***Complessità Naïf**: Esponenziale.*

4.1 Il Potere del Non-Determinismo

I problemi come SAT e Independent Set, pur essendo esponenziali per algoritmi deterministici naïf, diventano molto più "facili" se si introduce il non-determinismo.

Algoritmo Non-Deterministico per SAT:

1. **Fase di Guess (Indovina)**: Una Macchina di Turing non-deterministica "indovina" (o genera non-deterministicamente) un assegnamento di verità per le variabili della formula ϕ. Questo può essere fatto in tempo *lineare* ($O(n)$ dove n è il numero di variabili). Concettualmente, la macchina crea un branch per ogni possibile scelta di valore (vero/falso) per ogni variabile.
2. **Fase di Check (Verifica)**: Successivamente, la macchina verifica in modo deterministico se l'assegnamento indovinato soddisfa effettivamente la formula ϕ. Come visto, questa fase richiede tempo *polinomiale* nella lunghezza di ϕ.

Il computation time per questo algoritmo non-deterministico è la somma del tempo di guess e del tempo di check, risultando in un tempo *polinomiale*.

Algoritmo Non-Deterministico per Independent Set:

1. **Fase di Guess (Indovina)**: Una Macchina di Turing non-deterministica "indovina" un sottoinsieme di vertici $S \subseteq V$. Questo può essere fatto in tempo *lineare* ($O(V)$).
2. **Fase di Check (Verifica)**: La macchina verifica in modo deterministico se il sottoinsieme S indovinato è un independent set e se la sua dimensione è almeno k. Questo richiede tempo *polinomiale* (es. $O(V^2)$).

Anche in questo caso, il computation time è polinomiale.

Definizione 4.1 (Classe NTIME($t(n)$))**.** *Sia $t(n)$ una time function. La classe di complessità* ***NTIME($t(n)$)*** *è l'insieme di tutti i linguaggi L tali che esiste una Macchina di Turing* non-deterministica *M che decide L con un running time $O(t(n))$.*

La 'N' in NTIME sta per "Non-Deterministica".

Definizione 4.2 (Classe NP (Nondeterministic Polynomial Time))**.** *La classe* ***NP*** *è l'unione di tutte le classi NTIME(n^c) per ogni costante $c \geq 1$:*

$$NP = \bigcup_{c \geq 1} NTIME(n^c)$$

Nota Importante: NP sta per *Nondeterministic Polynomial* e non per "Non Polinomiale". La classe NP contiene tutti i problemi di decisione che possono essere risolti da una macchina di Turing non-deterministica in tempo polinomiale. Esempi di problemi in NP includono SAT, Independent Set, Knapsack, Hamiltonian Cycle, Traveling Salesperson Problem e molti altri problemi che sono centrali nell'informatica teorica e pratica.

Classe	**Tipo di Macchina**	**Running Time**
DTIME($t(n)$)	Deterministica	$O(t(n))$
P	Deterministica	$O(n^c)$ per qualche $c \geq 1$
NTIME($t(n)$)	Non-deterministica	$O(t(n))$
NP	Non-deterministica	$O(n^c)$ per qualche $c \geq 1$

La relazione tra P e NP è una delle questioni più importanti e irrisolte dell'informatica: $P \stackrel{?}{=} NP$.