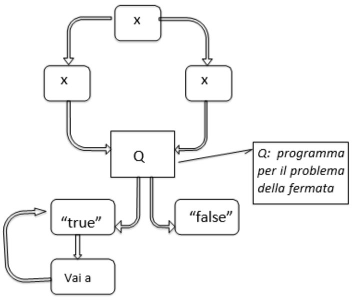
Valutazione dell'efficienza degli algoritmi: Bisogna tener conto: dell'elaboratore su cui il programma viene eseguito, del linguaggio di programmazione e dal compilatore utilizzato, dei dati di ingresso ai due programmi, della significatività dei dati di ingresso (è necessario eseguire i due programmi più volte con dati differenti).

Analisi asintotica e notazioni: Data una funzione $g(n)$ definita su N, $O(g(n))$ rappresenta l'insieme delle funzioni che per valori maggiori di n_0 assumono un valore pari al più a $c \cdot g(n)$. Osserviamo che se $f(n) \in O(n)$ allora abbiamo che $f(n) \in O(n^2)$ e $\in O(n^3)$. Data una funzione $g(n)$ definita su N, $\Omega(g(n))$ rappresenta l'insieme delle funzioni che per valori maggiori di n_0 assumono un valore almeno pari a $c \cdot g(n)$. Osserviamo che se $f(n) \in \Omega(n^2)$ allora abbiamo che $f(n) \in \Omega(n)$.

Equazioni di ricorrenza

$T(n) = T(n/2) + \Theta(1)$ e $T(1) = \Theta(1)$
$T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = 2 + T(n/4) = 2 + 1 + T(n/8) = 3 + T(n/8) = \ldots = T(n) = k + T(n/2^k)$
Assumiamo che n sia potenza di 2. In questo caso continuiamo a srotolare la ricorsione fin quando $n/2^k = 1$; ora $n/2^k = 1$ implica $2^k = n$ e quindi $k = \log_2 n$. Abbiamo quindi mostrato che la soluzione dell'equazione nel caso della ricerca binaria è $T(n) = \log_2 n + 1 = O(\log_2 n)$.

Problemi decidibili e indecidibili: Un problema di decisione T è decidibile se esiste un programma che termina sempre e riconosce le stringhe che verificano T, è semidecidibile se esiste un programma che termina sempre quando la risposta è true (non si richiede che il programma termini quando l'input non verifica la proprietà), è altrimenti indecidibile.



Il problema della fermata

Sia L un linguaggio di programmazione. Non esiste un programma Q scritto in L che, con input una coppia (x, y), dove x è la stringa che descrive un programma P scritto in L e y è una stringa di simboli di input per P, termina sempre in tempo finito e decide se P termina o no con input y. Dim: chiamo Q un programma che risponde vero o falso se il programma in input x termina o no. Chiamo Qmod il programma Q con l'aggiunta di un ciclo infinito se Q restituisce vero. Qmod termina dicendo falso se x cicla all'infinito. Quindi Qmod termina solo se x non termina. Passando Qmod a Qmod, questo terminerà solo se Qmod non termina!

La macchina di Turing universale: La macchina di Turing universale (U), proposta da Turing nel 1936, è una macchina di Turing in grado di simulare una qualunque altra macchina di Turing arbitraria su input arbitrari. La macchina universale prende in input sia la descrizione della macchina M da simulare che un input I e calcola il risultato di M con input I.

Tesi di Church-Turing: Tutto quello che si può calcolare lo si può calcolare con le macchine di Turing. In particolare, un problema di decisione è decidibile solo se esiste una macchina di Turing che riconosce il linguaggio associato e una funzione è calcolabile solo se esiste una macchina di Turing che la calcola.

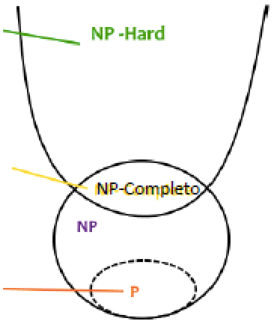
La classe P è l'insieme dei linguaggi L per i quali esiste una macchina di Turing con un solo nastro deterministica che decide L e per cui il numero di passi di calcolo su input di dimensione n è $O(n^k)$ per qualche $k \geq 0$ (polinomiale).

La classe NP è l'insieme dei linguaggi per cui esiste una macchina di Turing nondeterministica che in tempo polinomiale decide se x appartiene o meno a L. Ovviamente tutti i problemi che appartengono alla classe P appartengono anche a NP. $P \subseteq NP$ Non sappiamo se l'equivalenza precedente sia stretta o meno. Definizione alternativa: La classe NP è l'insieme di tutti i linguaggi L per cui, se x appartiene a L, allora esiste un certificato c(x) tale che c(x) ha lunghezza polinomiale in $|x|$ ed esiste una MdT deterministica che con input x e c(x) verifica che x appartiene a L in tempo polinomiale in $|x|$ e $|c(x)|$.

SAT: Il teorema SAT è NP-completo. Il problema SAT consiste nel decidere se una data formula booleana è soddisfacibile. Non si conoscono algoritmi polinomiali che risolvono questo problema, ma possiamo osservare che se una formula F con n variabili e m clausole è soddisfacibile, allora esiste un'assegnazione che soddisfa F. Un'assegnazione altro non è che una stringa di n valori binari, pertanto tale assegnazione può essere codificata con una stringa di lunghezza polinomiale nella lunghezza della codifica di F. Inoltre, esiste un semplice algoritmo polinomiale che verifica se un'assegnazione di valori di verità soddisfa la formula: tale algoritmo deve semplicemente sostituire alle variabili i valori assegnati e verificare che ogni clausola sia soddisfatta. Applicando questo algoritmo per ogni possibile combinazione avrà un costo esponenziale ($p(n)\,2^n$).

P è uguale a NP? Non abbiamo una dimostrazione che smentisca quest'affermazione ma pensiamo che la classe P sia inclusa propriamente nella classe NP. È come chiedersi se trovare una soluzione a un problema sia in generale più difficile o no che verificare se una soluzione è corretta. Se dimostrassimo che un problema NP-completo è risolubile in tempo polinomiale allora avremmo dimostrato anche che P=NP.

Riduzioni: Un linguaggio L1 è riducibile a un linguaggio L2 se esiste una funzione f: {0,1}* → {0,1}* detta riduzione, tale che, per ogni stringa binaria x, x appartiene a L1 se e solo se f(x) appartiene a L2. In questo caso se L2 è decidibile allora lo è anche L1. Se L1 non è decidibile allora non lo è neanche L2. L1 non è più difficile di L2 e L2 è almeno difficile quanto L1. Intuitivamente diciamo che L1 non è più difficile di L2, ovvero che se esiste una MdT che decide L2 allora ne esiste anche una che decide L1. La riduzione va dal problema che so indecidibile a quello che voglio dimostrare indecidibile.



Riduzioni polinomiali e NP-completezza: Un linguaggio L è NP-COMPLETO se appartiene a NP e se ogni altro linguaggio in NP è polinomialmente riducibile a L. Se un linguaggio L non è necessariamente NP ma ogni altro linguaggio in NP è polinomialmente riducibile a L allora è NP-difficile. Posso dimostrare che un linguaggio è NP-completo trovando una riduzione polinomiale da un altro linguaggio NP-completo (ad esempio SAT).

Riduzione da SAT a 3SAT: data una formula F con k clausole si definisce una formula F' che è soddisfacibile se e solo se F è soddisfacibile. Se k=1 mantengo solo la clausola l1 e tengo come problema D={{l1,y1,y2}, {l1,-y1,y2}, {l1,y1,-y2}, {l1,-y1,-y2}} questo è soddisfatto se e solo se l1 è soddisfatto. Se k=2 D={{l1,l2,y}, {l1,l2,-y}} che è soddisfatto se e solo se l1 e l2 sono soddisfatti. Se k>3 D={{l1,l2,y1}, {-y1,l3,y2}, {-y2,l4,y3}, {-y3,l5,l6}}.

Riduzione da 3SAT a Programmazione intera: se possiamo tradurre 3SAT in PL e sappiamo risolvere facilmente PL allora sappiamo anche risolvere 3SAT (falso). Ad ogni variabile di 3SAT associamo 2 variabili v e w e le limitiamo a 0 o 1 (falso o vero). Imponiamo che la loro somma sia 1 (uno vero e uno falso). Per ogni clausola di 3SAT definiamo un vincolo associato. Abbiamo riformulato 3SAT come PI e quindi possiamo dire che PI è un problema NP-difficile. Non abbiamo dimostrato che sia NP e quindi non possiamo definirlo NP-completo (sappiamo però che lo è).

Riduzione da 3SAT a 3-colorazione di grafi (è possibile colorare un grafo con 3 colori senza avere colori uguali adiacenti?). Costruiamo un grafo che simula un or e realizza una clausola. Alla fine, dimostriamo che decidere se un grafo è colorabile con k>3 colori è NP-completo. Con k=1 o k=2 è un problema risolubile in tempo polinomiale. Per alcune classi è più semplice (ad esempio un albero può essere colorato solo con 2 colori).

Linguaggi

Tipo di linguaggio	Tipo di produzioni	Modello di calcolo
Tipo 0	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$ e $\beta \in (V \cup T)^*$	Macchina di Turing
Contestuale	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$, $\beta \in (V \cup T)(V \cup T)^*$ e $\lvert\beta\rvert \geqslant \lvert\alpha\rvert$	Macchina di Turing lineare
Libero da contesto	$A \to \beta$ con $A \in V$ e $\beta \in (V \cup T)(V \cup T)^*$	
Regolare	$A \to aB$ e $A \to a$ con $A, B \in V$ e $a \in T$	Automa a stati finiti

I linguaggi di tipo 0 sono semi-decidibili mentre i linguaggi di tipo 1 sono decidibili.

E → E + E; E → E – E; E → E ∗ E; E → E/E; E → (E); E → id

E → E + T; E → E – T; E → T; T → T ∗ F; T → T/F; T → F; F → (E); F → id

Parser top-down: Un parser top-down parte dalla radice dell'albero di derivazione e cerca di ricostruire la crescita dell'albero che porta alla data sequenza di simboli terminali: nel fare ciò, ricostruisce una derivazione sinistra. Il parser top-down deve iniziare dalla radice dell'albero e determinare in base alla sequenza di simboli terminali come far crescere l'albero di derivazione: inoltre, deve fare questo in base solo alla conoscenza delle produzioni nella grammatica e dei simboli terminali in arrivo da sinistra verso destra. L'albero viene ricostruito in pre-ordine.

Perché una grammatica sia LL(1) richiediamo che per ogni produzione A → α | β, valgano le seguenti due proprietà: FIRST(α) − {λ} e FIRST(β) − {λ} siano disgiunti; se α è annullabile, allora FIRST(β) e FOLLOW(A) devono essere disgiunti.

Parser bottom-up: Generano derivazioni destre e costruiscono l'albero di derivazione partendo dalle foglie (in post-ordine). Si utilizza quindi l'analisi predittiva shift-reduce basata su grammatiche LR(k). Si leggono i token dall'input (shift) e si inseriscono in una pila tentando di costruire sequenze da ridurre con un non terminale (reduce). Se l'analisi va a buon fine si consuma tutto l'input ottenendo solo il simbolo iniziale nella pila. Se ad un certo punto non è più possibile operare uno shift o una reduce si ottiene un errore.