Complessità asintotica

Per poter valutare l'efficienza di un algoritmo, così da poterlo confrontare con algoritmi/programmi diversi che risolvono lo stesso problema, bisogna essere in grado di valutare l'ordine di grandezza delle sue necessità in termini di tempo di esecuzione e di memoria.

Tale valutazione è significativa se la dimensione dell'input è "sufficientemente" grande ovvero quando ha raggiunto una dimensione tale per cui una variazione non influisce più significativamente sulle misure.

Per questo motivo per quantificare tali grandezze ci si riferisce la complessità computazionale asintotica:

- *complessità computazionale* (o semplicemente *complessità*): indica che ci si occupa di valutare il "costo di esecuzione" di un algoritmo;
- *asintotica*: la valutazione avviene in situazion asintotica ovvero i risultati delle analisi sono validi per grandi dimensioni dell'input (che tendono all'infinito).

Due complessità abbiamo affrontato durante il corso. Complessità in **tempo** e Complessità in **spazio**

Per complessità in tempo si intende **NON** la quantità di memoria totale necessaria dall'algoritmo/programma per funzionare, **MA** il **massimo** numero di record di attivazione presneti sullo stack durante l'esecuzione.

Useremo la notazione matematica **O-grande** per descrivere la complessità asintotica delle varie funzioni/algoritmi/programma

In generale la notazione O-grande è utilizzata per descrivere il comportamento asintotico delle funzioni matematiche. Il suo scopo è descrivere come una funzione si comporta per valori elevati in modo e peremette di farlo allo stesso tempo in modo semplice e rigoroso.

Da un punto di vista formale:

date due funzioni $f(x)$, $g(x) \geq 0$ si dice che f(x) è un O(g(x))

se esistono due costanti c ed x_0 tali che $0 \leq f(x) \leq c\, g(x)$ per ogni $x \geq x_0$

Esempio

Ipotizziamo che il costo in tempo di una programma si descritto dalla $f(x) = (x + 1)^2$ (scritto in modo esteso da x^2+2x+1).

Quindi f (x) è $O(x^2)$, poiché se poniamo $c = 2$ e $n_0 = 3$ abbiamo che $(x + 1)^2 \leq 2x^2$, per ogni $n \geq 3$.

♦

Nota. Considerazione non-formale: essendo in considerazione il comportamento asintotico o all'infinito la funzione g(x) è definità usando solo l'elemento di grado maggiore in quanto gli altri elementi di un eventuale polinomio "perdono" significato sul comportamento all'infinito.

Quando un funzione ha un comportamento costante (non dipende dall'input ovvero l'input non influenza ciò che avviene come numero di operazioni, in pratica il codice porebbe essere scritto in modo statico senza aver necessità di costrutti di ciclo) si dice che la sua complessità è O(1).

Considerazioni sulla valutazione in tempo di un programma

Prima di cosa da definire è (la dimensione del)l'input che influenza l'esecuzione del programma da analizzare. Attenzione non tutti i parametri che sono passsati come input ad una funzione possono influenzare l'esecuzione del programma, per esempio alcuni possono essere utilizzati per restituire l'output.

Prima considerazione generale che si usa nella valutazione della complessità in tempo:

> ***la complessità di un programma è pari alla somma delle complessità delle istruzioni che lo compongono***

Possiamo avere diversi tipi di istruzioni:

- **le istruzioni elementari** (operazioni aritmetiche, lettura del valore di una variabile, assegnazione di un valore a una variabile, valutazione di una condizione logica su un numero costante di operandi, stampa del valore di una variabile, ecc.)
- l'istruzione ***if*** (condizione) ***then*** istruzione1 ***else*** istruzione2
- le **istruzioni iterative** (o iterazioni: cicli for, while)

Consideriamo il costo di ogni tipo di queste tipologie di operazioni:

- le istruzioni elementari hanno complessità ***costante*** ovvero O(1);
- l'istruzione *if (condizione) then istruzione1 else istruzione2* ha complessità pari al ***costo di verifica della condizione*** (tipicamente costante) più il costo delle operazioni collegate ovvero il ***massimo delle complessità di istruzione1 e di istruzione2***;
- le istruzioni iterative es. while(condizione) istruzione1 hanno una complessità che è cosi calcolata: ***costo*** (massimo) ***della singola iterazione*** moltiplicato per il **massimo numero di iterazioni possibili.** Per capire il numero di volte bisogna anlizzare la condizione del ciclo e le operazioni all'interno del ciclo che permettono di confutarla, con particolare attenzione se la condizione è composta (perché bisogna capire quale parte è la più difficile da confutare)

Attenzione quando si parla si parla si costo delle istruzioni bisogna distinguere tra le istruzioni di base o atomiche che hanno un costo costante e istruzioni non atomiche come il richiamo di una funzione, che di fatto "maschera" un pezzo di codice che ha una sua complessità che deve essere valutata e non è detto sia costante (dipende dal codice che la compone).

I vari tipi di istruzioni possono essere composte tra di loro, il modo più semplice è la sequenza, nel qual caso di *istruzione1; istruzione 2* si somma il costo delle varie istruzioni.

Di base is è mostrato i vari casi scrivendo istruzione1, che può far pensare ad una istruzione atomica. Si tratta di una astrazione, lì si potrebbe trovare quello che nel linguaggio di programmazione è considerato una istruzione. Per esempio nel ramo then di un if si potrebbe trovare un ciclo while e l'istruzione collega al while (ovvero il corpo del ciclo) potrebbe essere da una sequenza di istruzioni racchiuse tra parentesi graffe, delle quali una di esse è una ciclo for.

Sopra si è detto che il costo (o complessità) in tempo è pari alla somma del costo delle istruzioni che lo compongono. Questo calcolo è "immediato" in caso di una funzione iterativa, al contrario nel caso di una funzione ricorsiva bisogna calcolare il numero totale di chiamate ricorsive che verranno effettuate e moltiplicarlo per il costo del singolo passo di ricorsione.

Considerazioni sulla valutazione in spazio di un programma

Il costo per come lo abbiamo definito è una funzione matematica che descrive il numero **massimo** di record di attivazione presenti **contemporaneamente** sullo stack sulla base dell'input durante l'esecuzione del programma.

Se la funzione è *iterativa*, la sua complessità in spazio è O(1), in quanto è immediato affermare che ci sarà un solo record di attivazione presente nello stack e che quindi la sua complessità sia costante (ovvero non dipende dall'input).

Se la funzione è *ricorsiva*, bisogna analizzare i vari rami di ricosione per vedere come se si sviluppa.

Nel caso la funzione sia *ricorsiva lineare* (una sola chiamata ricorsiva per ogni flusso del codice che si sta analizzando) il numero massimo di record di attivazione coincide con il numero totale di record di attivazione allocati durante l'esecuzione. Questo si deve al fatto che ci sia una sola fase di crescita e una sola fase di decrescita (i.e. una volta arrivata al caso base/terminazione la funzione chiude a ritroso tutti gli step di ricorsione che erano attivi disallocando progressivamente i record di attivazioni sullo stack uno alla volta).

Quindi il valore che si andrà a calcolare in questo caso sarà utile per calcolare anche la complessità in tempo della funzione ricorsiva. Per questo motivo in caso di una funzione ricorsiva è consigliato partire dall'analisi della complessità in spazio.

Per fare questa analisi bisogna considerare come la funzione possa raggiungere il caso base o di terminazione: quindi prendere in analisi la condizione che lo indentifica e vedere come avvengono le varie chiamate ricorsive ovvero come i parametri delle varie chiamate ricorsive venogono aggiornati. L'attenzione si deve focalizzare sui parametri che sono oggetto delle condizioni del caso base. Nel caso si aggiornino anche altri parametri nelle chiamate a funzione questi parametri servono a memorizza parte della computazione effettuata e il loro aggiornamento non riguarda il raggiungimento della terminazione.

Prestare attenzione potrebbe esserci più di un un caso base o di terminazione. In questa situazione si deve identificare quello che comporta più lavoro ovvero più chiamtae ricorsivi (caso peggiore).

In molte situazione è possbile che ci siano rami del codice che non effettuano chiamate ricorsive, ma "semplicemnte" si occupano di calcolare e restituire il valore di output, perché sarebbe inutile continuare a lavorare una volta che è conosciuta la risposta. Esempio, nel caso delal ricerca di un valore nel momento in cui si trova quel valore, è inutile continuare ad analizzare anche i restanti perché la risposta sarà sempre "il valore è presente". Questi casi **non** sono casi base o di terminazione e non devono essere trattati come loro. Si tratta, infatti, di ottimizzazioni del codice al fine di fare, in alcune situzioni, meno lavoro, ma essendo interessati ai casi peggiori, dobbiamo cosiderare le situazioni in cui l'esecuzione del nostro programma non è avvantaggiata da queste ottimizzazioni ovvero si tratta di identificare il caso peggiore, quindi necessariamente non andrà a finire in quei rami del codice creati per limitare il lavoro computazionale (altrimenti non sarebbe il caso peggiore)