COMPLESSITÀ COMPUTAZIONALE

Un po' di teoria

La complessità computazionale analizza l'efficienza degli algoritmi, in termini di quantità di risorse necessarie per l'esecuzione dell'algoritmo. Le risorse tipicamente studiate sono memoria utilizzata e tempo impiegato. Noi ci focalizziamo solo sul tempo.

La complessità temporale si misura in n° di operazioni.

Poiché l'algoritmo è un metodo per risolvere un problema qualsiasi siano i dati, si vuole rende l'analisi di complessità generica, ovvero non dipendente da particolari dati del problema. Per questo si esprime la complessità in funzione della dimensione dei dati del problema. In genere si usa **n** per indicare la dimensione del dato.

Si usa la notazione teta grande (Θ), che può essere letto come "circa" o "segue", seguito dalla funzione a cui la complessità è asintotica.

Esempio: $\mathbf{\Theta(n^2)}$ significa che la complessità è "circa" $\mathbf{n^2}$ ovvero cresce come la parabola.

La complessità è sempre approssimata, questo la rende più facile da calcolare per problemi complessi e risulta anche più immediata. Le costanti si ignorano, quindi complessità **n** e complessità **n + 50** sono la stessa cosa. Questo perché la complessità si studia per problemi di grandissima dimensione. Quindi ogni costante, anche 100 o 1000, si può ignorare quando **n** è dell'ordine di $\mathbf{10^{10}}$ o più.

Infine la complessità si studia principalmente nel caso pessimo, ovvero il caso in cui l'algoritmo esegue il numero massimo di operazioni. Questo perché si vuole essere certi di quanto si debba aspettare al massimo perché l'algoritmo termini; se siamo fortunati e termina prima, buon per noi.

Alcune complessità tipiche, in ordine decrescente di efficienza:

$$1, \log(n), \sqrt{n}, n, n^2, n^3, ..., 2^n, 3^n, ...$$

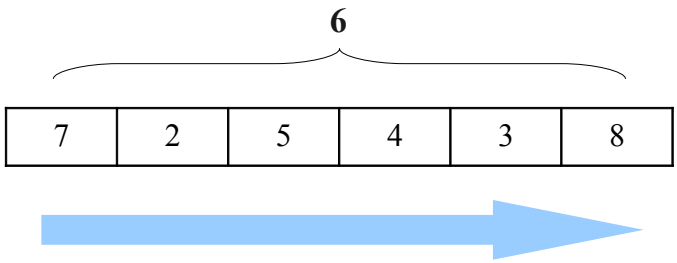
Qualche esempio:

- Svolgere una sola operazione qualsiasi sia il dato in ingresso: **Θ(1)**
- Svolgere un numero di operazioni uguali alla dimensione dell'ingresso: **Θ(n)**
- Svolgere un numero di operazioni uguali alla dimensione dell'ingresso per ogni elemento dell'ingresso: $\mathbf{\Theta(n^2)}$

Compito 1

Supponete di voler trovare la posizione di un particolare numero in una sequenza di numeri. La sequenza è data, mentre il numero da cercare è qualsiasi. L'algoritmo più efficiente consiste semplicemente nello scorrere tutta la sequenza in cerca del numero. Il caso pessimo si presenta quando il numero da cercare è in ultima posizione oppure non c'è: in quel caso eseguiamo tante operazioni (confronto col numero da cercare) quanti sono i numeri della sequenza. Perciò se la sequenza è lunga **n**, la complessità di tale algoritmo è **Θ(n)**.

Esempio:



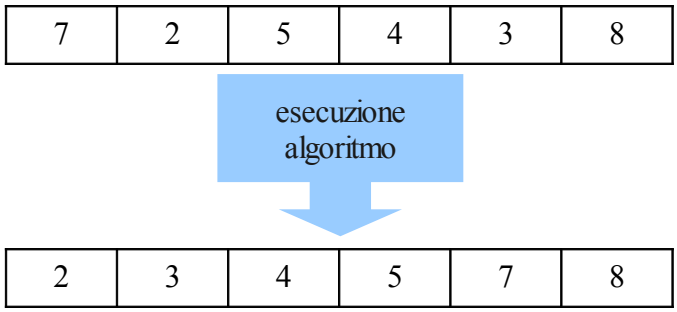
La sequenza è lunga 6. Se l'algoritmo deve cercare 7 farà una sola operazione di confronto con la prima cella (caso ottimo). Mentre se deve cercare 8 farà 6 confronti prima di trovarlo (caso pessimo).

Supponete ora invece che la sequenza sia già ordinata. Ovvero ogni elemento sia sempre minore di quello successivo (ignorate il caso in cui ci siano più elementi uguali).

Sfruttando questa proprietà, trovate un algoritmo di ricerca più efficiente, ovvero che faccia meno di **n** operazioni nel caso pessimo.

Compito 2

Trovate un algoritmo il più efficiente possibile che ordini una sequenza di numeri, forniti in ordine casuale. La complessità sarà sempre espressa in funzione della lunghezza **n** della sequenza.



Qualche consiglio

1. La complessità è una funzione di **n**. Perciò potete vederla in un grafico cartesiano, dove **n** è l'ascissa (x) e il numero di operazioni è l'ordinata (y). Più la funzione è ripida e peggio è l'efficienza. Nel caso di ricerca, se dovete migliorare il **Θ(n)**, l'obiettivo è arrivare ad una funzione meno ripida di **y = x**.
2. Per la ricerca pensate a quello che fate quando dovete cercare un numero di telefono sulla guida telefonica cartacea. Nonostante i numeri siano migliaia, sfruttate il fatto che siano ordinati in base al cognome per sfogliare solo un numero ridotto di pagine.
3. Per l'ordinamento potete usare tutta la memoria che volete per spostare i numeri o "parcheggiare" dei valori in attesa di posizionarli nella sequenza. Quindi concentratevi solo sul numero di confronti (=operazioni) da effettuare.
4. Chiaramente le sequenze mostrate sono solo esempi. L'algoritmo deve funzionare con qualsiasi sequenza di dati per cui si possa definire un ordine (non solo numeri). Inoltre pensate a sequenze molto grandi (miliardi di dati) e capirete perché ricerca e ordinamento sono così importanti.