Complessità computazionale

Definizione

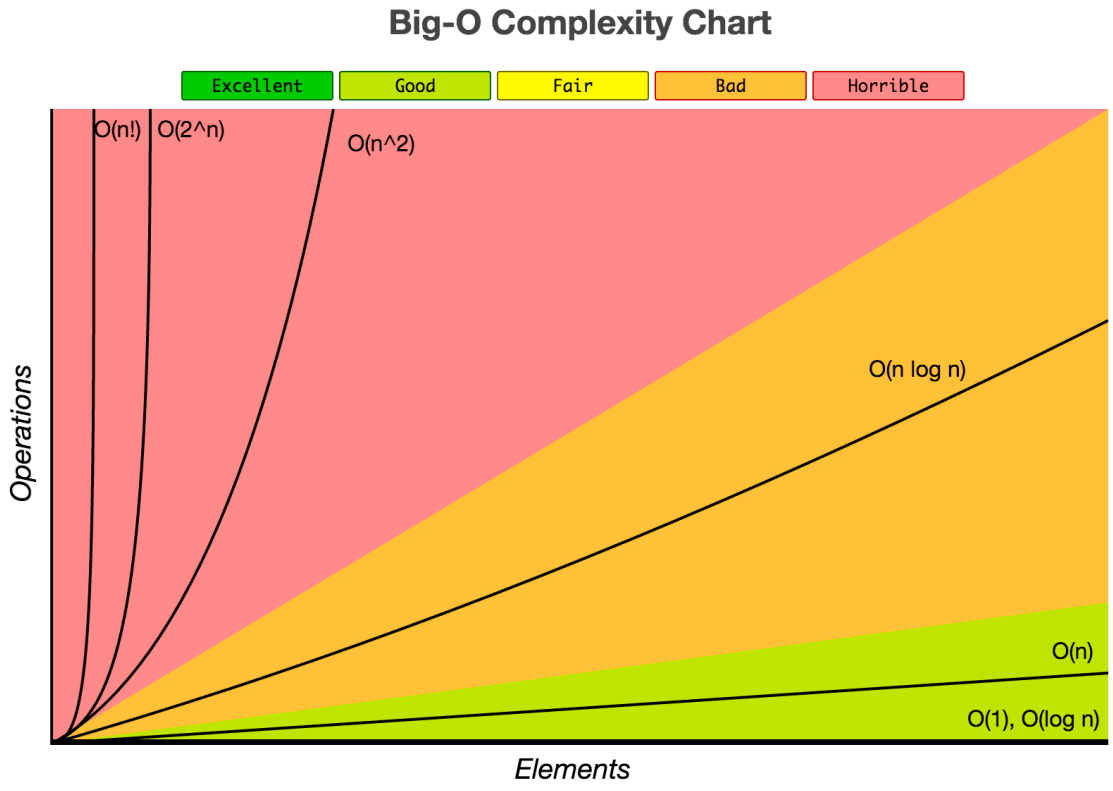
La complessità computazionale studia le risorse minime necessarie per la risoluzione di un problema. Con complessità di un algoritmo o efficienza di un algoritmo ci si riferisce dunque alle risorse di calcolo richieste. Per risorse di calcolo si intendono tempo e memoria utilizzati per l'esecuzione.

Notazione della O-grande

La notazione O-grande è un modo di descrivere la velocità o la complessità di un dato algoritmo. La notazione O-grande esprime il numero di operazioni compiute da un algoritmo.

> **ATTENZIONE!**
> La notazione O-grande non ti dice è la velocità di un algoritmo in secondi (ci sono troppi fattori che influenzano il tempo necessario all'esecuzione). Essa serve per confrontare algoritmi diversi tramite il numero di operazioni che svolgono.

Esempio:
Considera di impiegare 1 millisecondo per controllare ciascun elemento del database della scuola. Con una ricerca semplice devi controllare 10 voci ma con un algoritmo di ricerca binaria devi controllare soltanto 3 elementi. Nella maggior parte dei casi, la lista o il database in cui effettui la ricerca avranno centinaia o migliaia di elementi. Se c'è **1 miliardo** di elementi, usando la ricerca semplice il tempo massimo necessario è 1 miliardo di ms, o **11 giorni**. D'altro canto, usando la ricerca binaria ci vorranno soltanto **32 ms** nel caso più sfavorevole.
Chiaramente i tempi di esecuzione per una ricerca semplice e per una ricerca binaria non crescono alla stessa velocità!



Classi di complessità

Notazione	Nome	Esempio
$O(1)$	costante	Determinare se un numero è pari o dispari
$O(log(n))$	logaritmica	Ricerca binaria
$O(n)$	lineare	Ricerca lineare
$O(n \cdot log(n))$	loglineare	Merge sort, Heap sort
$O(n^2)$	quadratica	Insertion sort, Bubble sort
$O(n^k)$	polinomiale	Floyd-Warshall
$O(k^n)$	esponenziale	Trovare tutti i sottoinsiemi di un insieme
$O(n!)$	fattoriale	Trovare tutti i possibili ordinamenti di una lista

Esempi

Complessità dell'algoritmo	Tempo impiegato		
Mole di dati:	1 000	1 000 000	1 000 000 000
$O(log(n))$	0 s	0 s	0 s
Mole di dati:	1 000	100 000 000	1 000 000 000
$O(n)$	0 s	1 s	7 s
Mole di dati:	1 000	5 000 000	100 000 000
$O(n \cdot log(n))$	0 s	3 s	68 s
Mole di dati:	1 000	50 000	100 000
$O(n^2)$	0 s	10 s	48 s

Calcolo della complessità

Molti credono che la complessità indichi la velocità dell'algoritmo, ma questo non è corretto. La notazione Big O indica il "tasso di crescita" dell'algoritmo rispetto all'input.
Consideriamo, ad esempio, i 2 algoritmi in basso:

```
// Algoritmo 1
int n = 100;
for(int i = 0 ; i < n ; i++)
{
    cout<<"Ciao";
}
```

```
// Algoritmo 2
int n = 100;
for(int i = 0 ; i < n ; i++)
{
    cout<<"Ciao";
}
for(int i = 0 ; i < n ; i++)
{
    cout<<"Ciao2";
}
```

Ovviamente se eseguiamo i due algoritmi per lo stesso valore di n, l'algoritmo 1 impiegherà meno tempo rispetto all'algoritmo 2 per completare. Questo non significa che in termini di Big O i due algoritmi siano diversi. Molti penserebbero di usare la notazione O(n) per il primo algoritmo (dato che esegue n iterazioni) e O(2n) per il secondo algoritmo (dato che esegue le n iterazioni due volte). In realtà si sta semplicemente parlando di un algoritmo O(n). Questo è dato dal fatto che entrambi gli algoritmi scalano linearmente.

Tenendo in mente questo concetto è ovvio anche vedere perché i termini non dominanti vengono omessi nella notazione Big O. Per esempio consideriamo l'algoritmo in basso:

```
int n = 100;
for(int i = 0 ; i < n ; i++
{
    cout<<"Ciao";
}
for(int i = 0 ; i < n ; i++)
{
    for(int j = 0 ; j < n ; j++)
    {
        cout<<"Ciao2";
    }
}
```

A primo istinto verrebbe da dire che questo algoritmo è di tipo O(n + n^2) dato che prima esegue n iterazioni e poi n^2 iterazioni. Come prima abbiamo omesso la costante perché non necessaria per caratterizzare il tasso di crescita, anche qui possiamo fare lo stesso con n. La runtime di questo algoritmo è semplicemente O(n^2).