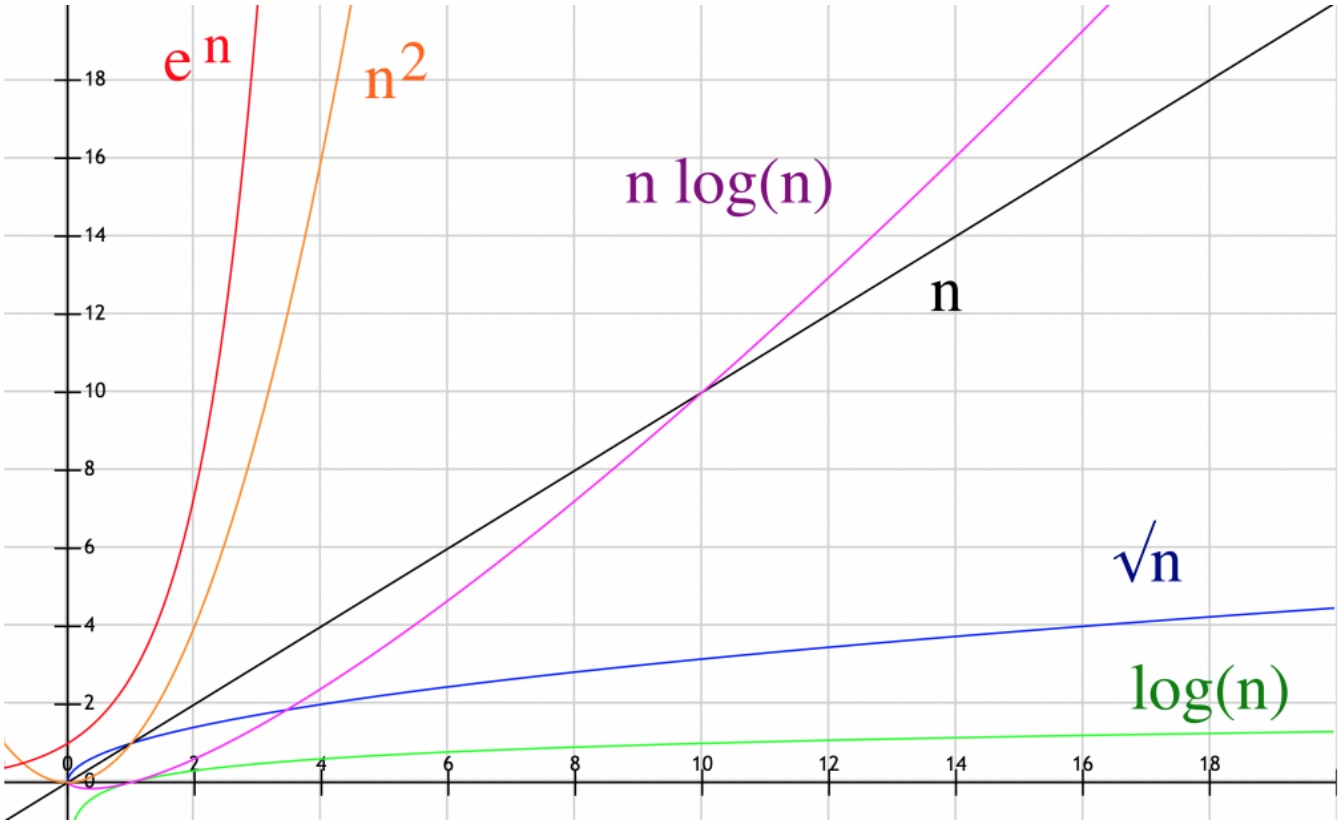
Complessità computazionale – Parte 1

Questo è il primo di una lunga (si spera) serie di articoli dedicata agli **algoritmi** ed **alle strutture dati**.
È doveroso fare una precisazione: l'articolo non vuole essere un sostituto dei più che validi testi dedicati all'analisi degli algoritmi (dei quali consiglio vivamente la lettura). Piuttosto vuole essere di facile accessibilità a tutti, anche a chi non ha basi matematiche solide e per chi vuole avvicinarsi all'argomento.
La **complessità computazionale** identifica le risorse minime necessarie (in termini di tempo di esecuzione e quantità di memoria utilizzata) per la risoluzione di un problema in funzione di un determinato tipo di *input*. Comprenderla appieno è fondamentale per uno sviluppatore perché consente di capire quando e quale algoritmo utilizzare e di giudicare quando un algoritmo diventa più *veloce* o più *lento*.
Gli argomenti trattati in questo articolo saranno:

ordini di funzione

Nell'analisi degli algoritmi si è soliti utilizzare le seguenti classi di funzioni (in ordine crescente di grandezza):

- Funzione costante: **O(1)**
- Funzione logaritmica: **O(*log*(n))**
- Funzione lineare: **O(n)**
- Funzione loglineare: **O(n *log(n))***
- Funzione quadratica: **$O(n^2)$**
- Funzione polinomiale: **$O(n^c)$**
- Funzione esponenziale: **$O(c^n)$**
- Funzione fattoriale: **O(n!)**



spazio di esecuzione

Con questa metrica si indica lo spazio di memoria occupato da un algoritmo. Ad esempio, se si vuole creare un *array* di *n* elementi, l'algoritmo richiederà uno spazio pari a **O(n)**. La creazione di una *matrice* di dimensione *n x n* richiederà uno spazio pari a **O(N²).** L'algoritmo di *ricerca lineare* in un array di dimensione *n*, richiederà uno spazio pari a **O(1)**, poiché non utilizza *strutture dati* **supplementari** per la sua esecuzione.

Tempo di esecuzione

Big O, Big Theta e Big Omega

Vengono utilizzate tre *notazioni* per indicare il tempo di esecuzione di un algoritmo. Ne segue una descrizione molto breve, senza scendere nei numerosi dettagli matematici:

- **O (big O – O grande):** indica il limite superiore della

funzione asintotica e può essere considerata come la relazione matematica *minore o uguale*. Ad esempio, l'*iterazione* di tutti gli elementi di un array di dimensione pari ad *N*, avrà un tempo di esecuzione pari a **O(N)**, poiché si ha la necessità di iterare tutti gli *N* elementi. Il tempo di esecuzione dell'esempio precedente può essere indicato anche come **O(N²)**, oppure come **O(N³)**, poiché è certo che l'algoritmo avrà un tempo di esecuzione *minore o uguale* a questi valori.

- **Ω (big Omega – Omega grande):** indica il limite inferiore della funzione asintotica e può essere considerata come la relazione matematica *maggiore o uguale*. Sempre in riferimento al tempo di esecuzione dell'esempio precedente, si indica come **Ω(N)** oppure **Ω(1)**, perché si è sicuri che l'algoritmo **non** possa essere **più veloce** di tale valore.
- **0 (big Theta – relazione Theta):** indica che il tempo di esecuzione di un algoritmo è sia **O** che **Ω**. Quindi, il tempo di esecuzione sarà pari a **0**(N) se tale algoritmo è sia **O(N)** che **Ω(N)**.

Solitamente viene utilizzata l'analisi **Big O** come riferimento per l'analisi della complessità computazionale. Negli esempi che seguono ci si riferirà proprio ad essa.

Caso migliore, caso peggiore e caso medio

Possiamo descrivere il tempo di esecuzione di un algoritmo in tre diversi modi. Per comprenderli al meglio, consideriamo come esempio la *ricerca sequenziale* in un array.

- **Caso migliore:** l'elemento ricercato è il primo dell'array;
- **Caso peggiore:** l'elemento ricercato è l'ultimo dell'array o non è presente;
- **Caso medio:** ciascun elemento ha la stessa probabilità di essere trovato;

Solitamente il tempo di esecuzione viene valutato nel caso *peggiore* o in quello *medio*. Spesso il caso medio e quello peggiore sono gli stessi.

CICLI

La complessità computazione di un ciclo *for* , *while* o *do-while* è data dal prodotto della complessità del corpo del ciclo stesso (**O(f(n)**) per il numero di volte che esso viene eseguito (**O(k)**). Nel caso del ciclo *for* la complessità delle operazioni di *inizializzazione*, *confronto* ed *incremento* è pari ad **O(1)** quindi *trascurabile*. In definitiva, il tempo di esecuzione è pari a:

O(k * f(n))

eliminare le costanti

È opportuno rimuovere i valori costanti dal tempo di esecuzione. Ad esempio, un algoritmo che potrebbe essere descritto con una complessità computazionale pari a **O(2N)**, viene indicato come **O(N)**. Si consideri un semplice algoritmo di ricerca dei valori *minimo* e *massimo* in un array.

```
int min = array[0];
int max = array[0];
for (int i=1; i<array.length; i++) { if (array[i] < min) min = array[i]; if (array[i] > max) max = array[i];
}
```

```
int min = array[0];
int max = array[0];
for (int i=1; i<array.length; i++) { if (array[i] < min) min = array[i];
}
for (int j=1; j<array.length; j++) { if (array[j] > max) max = array[j];
}
```

Il tempo di esecuzione di ambedue gli algoritmi è **O(N)**. Nel

secondo esempio vengono utilizzati due cicli for anziché uno, ma è errato indicare il tempo di esecuzione come **O(2N)**, perché andrebbe fatta un'analisi a basso livello di come il compilatore effettuerà le varie ottimizzazioni e tutta un'altra serie di dettagli per comprenderne il *reale* tempo di esecuzione. La **Big O** ci permette di eliminare le *costanti* e di darci un'idea di come il tempo di esecuzione scali.

eliminare i termini non dominanti

Così come per i valori costanti, bisogna eliminare i termini non dominanti, poiché non sono particolarmente importanti ai fini dell'analisi **Big O**. Ad esempio:

- **$O(N^2+N)$** diventa **$O(N^2)$**;
- **$O(N^2+N^2)$** diventa **$O(N^2)$**;
- **$O(N^2+N)$** diventa **$O(N^2)$**;
- **$O(N + \log N)$** diventa **$O(N)$**;

OPERAZIONI PRIMITIVE

Le operazioni primitive corrispondono ad istruzioni di alto livello che possono essere eseguite con poche istruzioni di linguaggio macchina. Di conseguenza il loro tempo di esecuzione è pari a **O(1)**. Alcuni esempi sono:

- Assegnazioni di valori (*int a=0*, *String s = "a"*, ….);
- Operazioni aritmetiche (+, –, *, /);
- Operazioni logiche (&, &&, |, ||);
- Confronti (<, <=, >, >=, ==);
- Accesso ad un elemento di strutture dati semplici, ad esempio un array;
- Operazioni di controllo (*return, break, continue*);

Costrutto if-else

La complessità computazione di un costrutto *if-else* è determinata dalla complessità maggiore tra quella dell'*if*

(**O(f(n)**) e quella dell'*else* (**O(g(n)**). Il *confronto*, solitamente, ha una complessità pari a **O(1)**. In definitiva, il tempo di esecuzione è pari a:

O(max[f(n), g(n)])

chiamata di funzione

La complessità computazionale di una *chiamata di funzione* è data dalla somma del costo di esecuzione della funzione stessa (**O(f(n)**) ed il costo della chiamata. Quest'ultimo è in genere trascurabile, ad eccezione dei casi in cui viene passata una *struttura dati* come parametro ed il linguaggio di programmazione in uso crea una copia della stessa (**O(g(n)**). In definitiva, il tempo di esecuzione è pari a:

O(f(n) + g(n))

addizione vs moltiplicazione

Quando ci si trova davanti a due cicli for si fa confusione sul quando *moltiplicare* i due tempi di esecuzione e sul quando *sommarli*. Se i due cicli for sono annidati, bisognerà **moltiplicare** i tempi di esecuzione:

```
for (int i=0; i<arrA.length; i++) { for (int j=0; j<arrB.length; j++) { print(arrA[i] + " - " + arrB[j]); }
}
```

Il tempo di esecuzione del precedente esempio è pari a **O(A*B)**.

Diversamente, se i due cicli for sono sequenziali, bisognerà **sommare** i tempi di esecuzione:

```
for (int i=0; i<arrA.length; i++) { print(arrA[i]);
}
for (int j=0; j<arrB.length; j++) { print(arrB[j]);
}
```

Il tempo di esecuzione del precedente esempio è pari a **O(A+B)**.

TEMPO AMMORTIZZATO

Alcune strutture dati necessitano di *espandersi* nel momento in cui sono piene. Un esempio nel linguaggio Java è l'*ArrayList*: non appena si raggiunge la sua massima capacità, viene creato un nuovo array con capacità doppia e vengono copiati tutti gli elementi dal vecchio array a quello nuovo. Se il vecchio array conteneva *N* elementi, questa operazione richiederà un tempo pari a **O(N)**. Nonostante ciò, la maggior parte degli inserimenti avverrà in un tempo pari ad **O(1)**, questo perché si considera il **tempo ammortizzato**. Questo concetto serve a descrivere il fatto che il **caso peggiore** può capitare, ma ciò avviene raramente, di conseguenza il costo è "**ammortizzato**".

log n

Spesso si notano valori di **O(log N)** nei tempi di esecuzione degli algoritmi. Per identificare semplicemente quest'ordine di funzione bisogna notare se il problema ha un numero di elementi che viene dimezzato ogni volta; in tal caso, molto probabilmente, ci troveremo di fronte ad un tempo di esecuzione pari a **O(log N)**.

Un esempio si ha nell'algoritmo di *ricerca binaria*: ad ogni iterazione dimezziamo il numero di elementi dell'array da esaminare.

conclusioni

Come già detto ad inizio articolo, la *complessità computazionale* è l'argomento cardine dell'analisi degli algoritmi, che qualsiasi sviluppatore dovrebbe conoscere per scrivere un buon ed efficiente codice. Se a primo acchito possa risultare un argomento di difficile comprensione, è necessario solo un po' di esercizio nell'identificare i tempi di esecuzione dei vari algoritmi. Fortunatamente la notazione **Big O** rende ancora più facile questo compito grazie

alle sue poche e semplici regole.

bibliografia

Cracking the Coding Interview di *Gayle Laakmann McDowell*.

https://italiancoders.it/complessita-computazionale-parte-1/