9. ANALISI DEGLI ALGORITMI

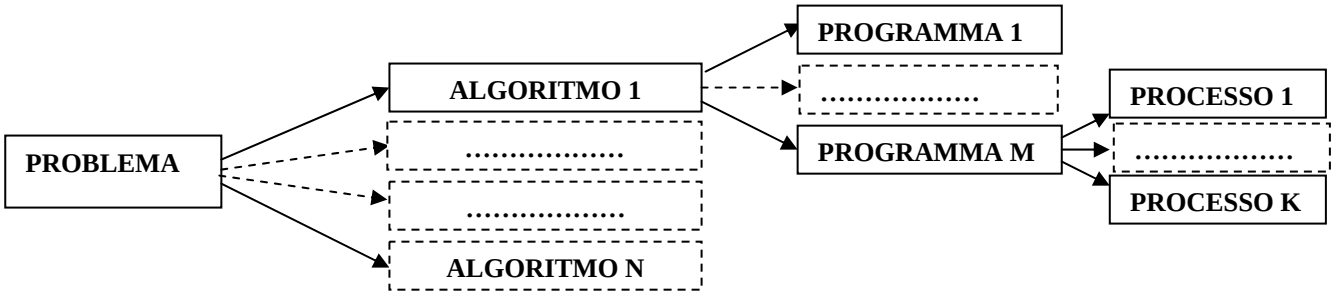
Abbiamo visto che in informatica esistono molte strade per risolvere uno stesso problema ossia esistono in genere più processi risolutivi equivalenti in grado a partire dagli stessi dati di input di ricavare gli stessi dati di output.
Quindi in informatica ci si occupa spesso durante la fase di analisi, per quanto possibile, di trovare *"buone soluzioni"* di un problema ossia ci si sforza di trovare soluzioni quanto più *"economiche"* possibile in termini di *"efficienza"* in grado di risparmiare per quanto possibile *"tempo di esecuzione"* e *"spazio di memoria"*.

Domanda chiave: Quali sono i criteri da seguire per valutare la bontà di un algoritmo?

Rispetto ad un qualsiasi algoritmo dobbiamo considerare i seguenti **due aspetti**:

a) la sua **organizzazione interna**: ossia l'organizzazione delle sue strutture di controllo e delle strutture dati utilizzate;
b) **le risorse necessarie** per eseguirlo: (in particolare la memoria ed il processore) strettamente legate allo spazio occupato ed al tempo di esecuzione.

Ricordiamo che ogni **algoritmo** che risolve un determinato **problema** può essere tradotto in un **programma** scritto in un qualsiasi linguaggio di programmazione e che tale programma verrà trasformato in un **processo** a tempo di esecuzione.



Per valutare oggettivamente la **bontà di un algoritmo** non ci riferiremo direttamente all'algoritmo in quanto tale, bensì alle risorse (tempo di esecuzione e spazio di allocazione) che utilizzerà quando verrà trasformato in processo ossia in *entità dinamica a tempo di esecuzione.*

La parte dell'informatica che si occupa dello studio e dell'individuazione della complessità di calcolo o computazionale di un algoritmo si chiama **analisi computazionale**.
La conoscenza dei principi di tale disciplina ci permette di ampliare il nostro obbiettivo di programmazione passando dall'iniziale:
"dato un problema trovare un algoritmo corretto e funzionante che ne descriva il relativo procedimento risolutivo e codificarlo in un determinato linguaggio di programmazione (per noi il C)"

al più completo:

"dato un problema trovare tra gli algoritmi risolutivi possibili quello migliore confrontandoli dal punto di vista dell'efficienza sulla base di un analisi qualitativa delle sue istruzioni"

Siamo quindi interessati costruire algoritmi che abbiano una organizzazione interna tale da *minimizzare* le risorse utilizzate ossia:

1) **spazio di memoria**: ossia l'area di memoria (di lavoro) occupata da un *processo* durante la sua esecuzione riferendoci sia a quella necessaria per memorizzare le strutture dati utilizzate sia a quella necessaria per memorizzare il codice stesso, i suoi dati di input e di output ed i risultati intermedi;

2) **tempo di esecuzione**: ossia il tempo di esecuzione legato al processo necessario alla sua esecuzione.

Tra le due risorse in mancanza di esplicita richiesta si considera prioritaria la risorsa "tempo di esecuzione" pertanto nel prosieguo della nostra analisi valuteremo esclusivamente il tempo di esecuzione di un processo legato ad un algoritmo.
Per far ciò **confronteremo** tra loro due o più algoritmi equivalenti di una stessa classe di problemi sulla base degli stessi **parametri** in modo da potere effettuare **un'analisi qualitativa** degli stessi.

UN MODO **NON ADATTO** PER VALUTARE IL TEMPO DI ESECUZIONE DI UN ALGORITMO

Ipotizziamo di **valutare il tempo di esecuzione** di un algoritmo esprimendo il tempo nelle **unità solari** abituali (cronometro con ore, minuti e **secondi**).
Per valutare quindi la bontà di un certo processo risolutivo relativo ad un determinato algoritmo è sufficiente cronometrarne il tempo impiegato per la sua esecuzione. Analogamente si fa per tutti i processi risolutivi **equivalenti** individuati. Secondo questo criterio confrontando semplicemente i tempi misurati sarà possibile individuare l'algoritmo più efficiente.

Questo metodo, apparentemente corretto, è assolutamente inaccettabile!!!

I risultati del test proposto sono **inattendibili** in quanto verranno a dipendere esclusivamente dalle condizioni in cui verrà eseguito il test e precisamente:
- dalla **velocità di esecuzione** dell'elaboratore;
- dalla **velocità d'interpretazione** del programma traduttore (compilatore o interprete) utilizzato per la codifica;
- dalla **dimensione** e dalla **disposizione** dei dati di test forniti in input.

Appare evidente che un confronto può essere ritenuto valido solo se i diversi processi in esecuzione vengono testati nelle stesse condizioni ossia:
- i diversi programmi sono tradotti utilizzando **lo stesso traduttore** (compilatore o interprete);
- il test viene eseguito **sullo stesso elaboratore** dedicato;
- il test viene eseguito più volte con dati di input diversi per **dimensione** e **disposizione**.

Queste osservazioni ci portano ad affermare che in ogni caso non si può assolutamente valutare la complessità di un algoritmo servendosi dell'unità solari in quanto nonostante tutti gli accorgimenti più minuziosi che è possibile prendere, non può essere comunque effettuata un'analisi completa.

Per questo motivo si dice che per valutare correttamente la bontà di un algoritmo occorre considerare esclusivamente **l'algoritmo in se** (i dati e le istruzioni utilizzate) e **non la sua implementazione.**

IL MODO **ADATTO** PER VALUTARE IL TEMPO DI ESECUZIONE DI UN ALGORITMO

Un metodo più idoneo di valutazione del *tempo di esecuzione di un algoritmo* è quello di esprimerlo in funzione del **numero di operazioni** (assegnazioni, confronti, operazioni, di I/O, operazioni aritmetiche, scambi, etc.) che l'algoritmo deve compiere per fornire i risultati.

Definiremo **costo di un algoritmo** il numero di operazioni necessarie a realizzare il suo processo risolutivo

Introdurremo poi le seguenti regole di valutazione per il calcolo del costo di un algoritmo in un linguaggio strutturato:

1) Vanno considerate solo le istruzioni comprese tra **INIZIO** e **FINE** senza includere queste due istruzioni nel calcolo relativo;
2) Le **istruzioni semplici** quali lettura, scrittura, assegnazione, test, hanno un costo pari ad 1;
3) Le **istruzioni iterative** (quali **MENTRE**…FINE MENTRE, **RIPETI**..FINCHE', **PER**…FINE PER) hanno un costo pari alla somma dei costi del test e del corpo del ciclo. In particolare:
   - il *test del ciclo* essendo un'istruzione semplice ha costo uguale ad 1 per il numero di volte che viene eseguito il ciclo;
   - il corpo del ciclo avrà costo pari alla somma dei costi delle singole istruzioni tenendo conto di quante volte vengono eseguite.
4) **L'istruzione di selezione unaria SE**…**ALLORA** ha costo pari alla somma del costo del test ossia 1 più quello delle singole istruzioni contenute nel ramo ALLORA. Analogamente viene calcolato il costo **dell'istruzione di selezione binaria SE**…**ALLORA** …**ALTRIMENTI** che ha costo pari alla somma del costo del test ossia 1 più, per convenzione, quello **massimo** tra i costi delle singole istruzioni contenute nel ramo ALLORA e del ramo ALTRIMENTI. Analogamente viene calcolato il costo **dell'istruzione di selezione ennaria NEL CASO CHE ….** FINE CASO che ha costo pari alla somma del costo del test ossia 1 più, per convenzione, quello **massimo** tra i costi dei vari blocchi relativi alla lista dei valori oppure al ramo ALTRIMENTI.
5) L'**istruzione di chiamata di un sottoprogramma** (funzione o procedura) ha ovviamente costo pari a quello dell'intero sottoprogramma tenendo conto delle regole finora elencate più il costo dell'istruzione di chiamata che per convenzione è pari ad 1;
6) **L'istruzione composta** (ad esempio strutture di controllo annidate e blocchi di istruzioni) ha una complessità pari alla somma dei costi delle singole istruzioni semplici che la compongono

I risultati ottenuti in accordo alle seguenti regole sono più affidabili rispetto ai precedenti benché risultino anch'essi **approssimati per difetto** in quanto è consuetudine occuparsi esclusivamente delle operazioni dal costo dominante (confronti e moltiplicazioni) trascurando le istruzioni meno onerose dal punto di vista esecutivo quali addizioni, sottrazioni, incrementi, decrementi.

Possiamo allora finalmente dire che **il tempo di esecuzione di un qualsiasi algoritmo è proporzionale al suo costo.**
Infatti appare evidente che conoscendo il tempo di esecuzione di una istruzione di costo unitario (**tempo unitario**) espresso in secondi basterà moltiplicare il costo complessivo dell'algoritmo per il tempo unitario per ottenere **il tempo totale di esecuzione** dell'algoritmo.

Esempio: calcolare il costo del seguente algoritmo:

```
ALGORITMO A
i, p, som: INT

INIZIO
Scrivi("Inserisci un numero")
Leggi (p)
i ← 1
som ← 0
MENTRE (i<= p) ESEGUI
  som ← somma (i, som)
  i ← i + 1
FINE MENTRE
Scrivi ("La somma richiesta è")
Scrivi (som)
FINE
```

```
FUNZIONE somma (VAL x: INT; VAL y: INT): INT
s: INT

INIZIO
s ← x + y
RITORNA (s)
FINE
```

: INT

Supponiamo che **p = 10** allora

costo A = 5 + (1 +1 +1 +costo chiamata funzione somma( ) + costo funzione somma ( )) * 10) + 1 (ultimo test) = **66** perché il costo della funzione somma( ) è pari a 2

in generale per un p generico

costo A = 5 + (1 +1 +1 +costo chiamata funzione somma( ) + costo funzione somma ( )) * p) + 1 (ultimo test) = **6p + 6**

OSS: il numero di operazioni è funzione della dimensione dei dati di input

Esempio: calcolare il costo del seguente algoritmo:

```
ALGORITMO B
x, y, r, s, z :INT

INIZIO

SE (x < 0)
  ALLORA
    SE (y < 0)
      ALLORA
         r ← 1
      ALTRIMENTI
         r ← 2
         s ← 1
    FINE SE
    z ← 1
FINE SE

FINE
```

costo B = 1 + costo del SE interno +1 = 1 + (1 + 2) +1 = **5**

DIMENSIONE DEL PROBLEMA E COMPLESSITA' COMPUTAZIONALE

Il numero di operazioni di un algoritmo come abbiamo potuto notare è legato **alla dimensione dei dati di input.** Quindi possiamo definire il numero di operazioni in funzione della dimensione dei dati di input.

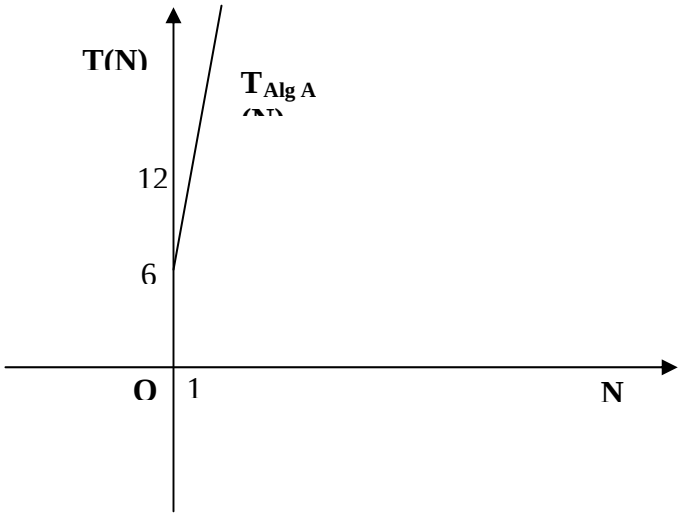
Si usa perciò la notazione generica **T(N)** per indicare la **funzione matematica** che indica la relazione esistente tra il numero di operazioni di un programma e la dimensione N dei dati in input.

OSS: Per maggiore chiarezza utilizzeremo **N** nei casi generici (e nelle definizioni) e **P** negli algoritmi e negli esempi.

La **funzione T(N)** è chiamata **complessità computazionale** (*in tempo*) o semplicemente **complessità** di un algoritmo.
La **dimensione N** dei dati in input è anche detta **dimensione del problema.**

Essendo T(N) una funzione matematicaè possibile rappresentarla attarevsro il suo grafico in un sistema di assi cartesiano ponendo
sull'asse delle ascisse o asse x : la dimensione del problema N
sull'asse delle ordinate o asse y : la complessità del problema ossia il numero di operazioni T(N).

Esempio: costo dell'algoritmo A calcolato in precedenza ***6p + 6***



Esempio: commentare e descrivere gli esempi di calcolo del costo degli algoritmi a pag 18 – Paragrafo 5

ANALISI AL CASO OTTIMO, PESSIMO E MEDIO

Il termine **complessità** qui introdotto risulta ancora molto generico ed impreciso.
E' importante infatti valutare la sequenza di dati che si presentano in input ed i loro specifici valori.
La **complessità** di un algoritmo pertanto è legata ai valori dati in input ed **a come i dati sono disposti oltre che alla loro dimensione**.

In definitiva per valutare le prestazioni di un algoritmo si dovrebbero prevedere tre tipi di analisi ossia:

a) analisi al caso **ottimo**;
b) analisi al caso **pessimo**;
c) analisi al caso **medio**.

La funzione complessità va analizzata nel caso più favorevole,in quello sfavorevole, e nel caso medio in base alla disposizione ed al tipo dei dati in ingresso. Per esprimere tali funzioni di complessità utilizzeremo le seguenti notazioni:

a) $\mathbf{T_{ottimo}(N)}$;
b) $\mathbf{T_{pessimo}(N)}$;
c) $\mathbf{T_{medio}(N)}$.

Per decidere quale considerare delle tre, va analizzato il tipo di applicazione, ma in genere è la funzione $T_{medio}(N)$ quella che riveste la maggiore importanza. Se risultasse difficile da individuare ci si riferirà alla funzione $T_{pessimo}(N)$.

```
FUNZIONE Ric_Seq (VAL v ARRRAY[ ] DI INT,
                  VAL dim: INT,
                  VAL x: INT): BOOL
trovato: BOOL
i: INT

INIZIO
trovato ← FALSO
i ← 1

MENTRE (i<= dim) AND (NOT trovato) ESEGUI
  SE (v[i] = x)
    ALLORA
      trovato ← VERO
  FINE SE
  i ← i + 1
FINE MENTRE

RITORNA (trovato)

FINE
```

```
FUNZIONE Ric_Bin (VAL v ARRRAY[ ] DI INT,
                  VAL dim: INT,
                  VAL x: INT): BOOL
trovato: BOOL
primo, ultimo, centro: INT

INIZIO
primo ← 1
ultimo ← dim
trovato ← FALSO

MENTRE (primo <= ultimo) AND (NOT trovato) ESEGUI
  centro ← (primo + ultimo) DIV 2

  SE (v[centro] = x)
    ALLORA
      trovato ← VERO
    ALTRIMENTI
      SE (v[centro] < x)
        ALLORA
          primo ← centro + 1
        ALTRIMENTI
          ultimo ← centro – 1
       FINE SE
  FINE SE

FINE MENTRE

RITORNA (trovato)

FINE
```

Sottoponiamo l'algoritmo **di ricerca sequenziale** sul vettore ai tre tipi di analisi descritte e soffermiamoci sulla ricerca con successo di un valore:
- caso ottimo: il valore x da ricercare è all'inizio del vettore ed il numero di confronti da effettuare è pari ad **1**;
- caso pessimo: il valore x da ricercare è alla fine del vettore ed il numero di confronti da effettuare è pari alla lunghezza del vettore **dim**;
- caso medio: il valore x da ricercare è a metà del vettore ed il numero di confronti è pari circa a **dim/2** se il valore da ricercare +è scelto in modo casuale;

Sottoponiamo l'algoritmo di ricerca binaria sul vettore ai tre tipi di analisi descritte e soffermiamoci sulla ricerca con successo di un valore:
- caso ottimo: il valore x da ricercare è proprio a metà del vettore ed il numero di confronti da effettuare è pari ad **1**;
- caso pessimo: si ha quando è necessario suddividere il vettore V fino a ridurlo ad un solo elemento. Nel caso pessimo occorrono **log dim** confronti;
- caso medio: l'analisi è notevolmente complessa e riportiamo solo il risultato finale sul numero di confronti che è **log dim**;

Tabella riassuntiva comparativa

	Ricerca sequenziale	**Ricerca binaria**
Caso ottimo	**1**	**1**
Caso pessimo	**dim**	**log dim**
Caso medio	**dim / 2**	**log dim**

Come è possibile osservare gli algoritmi hanno la stessa complessità nel caso ottimo, ma complessità diverse nel caso medio e nel caso pessimo.
L'obiettivo di un buon programmatore sarà quello di risolvere un problema scrivendo un algoritmo con la **minima complessità** possibile *sia al caso medio sia a quello pessimo*: definiremo tale algoritmo **un algoritmo ottimo.**

Pertanto per **complessità di un problema** intenderemo la **complessità degli algoritmi ottimi** che risolvono quel problema (e spesso tale complessità è già nota a priori).

ORDINE DI GRANDEZZA E CLASSI DI COMPUTABILITA'

Confrontando 2 algoritmi può accadere che il primo esegua meno operazioni del'altro quando la dimensione del problema è bassa, ma le cose si ribaltino quando la dimensione cresce.

Quindi conviene fare riferimento **all'ordine di grandezza delle complessità** ossia valutare la complessità per valori molto grandi delle dimensioni del problema.
Si parla in questo caso di **complessità asintotica** e si esprime in notazione matematica nel seguente modo

$$\lim_{N \to \infty} \mathbf{T(N)}$$

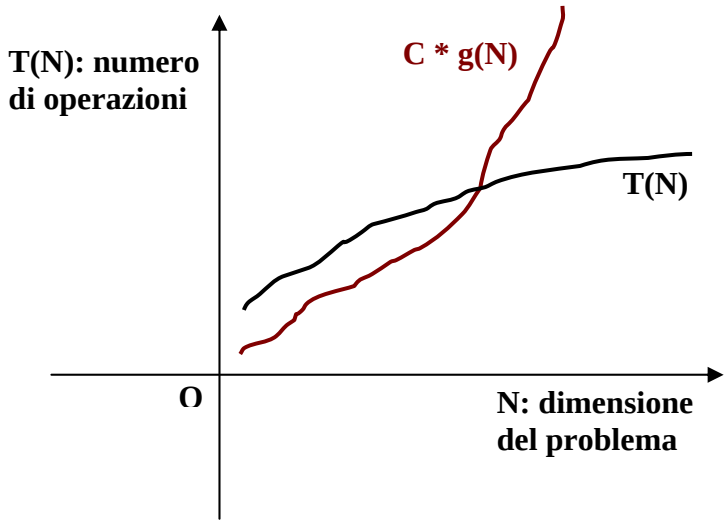
In altre parole si ottiene una espressione in funzione di N che indica qual è il comportamento all'infinito (asintotico) dell'algoritmo.

Introduciamo le seguenti definizioni:

Siano **f(N)** e **g(N)** due funzioni.
Si dice che **f(N) è di ordine di grandezza g(N)** e si scrive **O(g(N))** e si legge *"O grande di g di N"* se esiste una costante **C > 0** tale che per tutti i valori **N > 0** accade che

f(N) < C * g(N)

Pertanto dire che T(N) è O(g(N)) significa dire che g(N) è un limite superiore alla crescita di T(N) ossia che T(N) non cresce più di g(N) ed il suo grafico da un certo punto in poi sarà sotto al grafico di g(N).



Si dice che **f(N) è proporzionale a g(N)** ossia che **f(N) e g(N) sono dello stesso ordine di grandezza** se

f(N) è O(g(N)) e **g(N) è O(f(N))**

Sarà quindi possibile fare affermazioni del tipo "la complessità computazionale T(N) per questo algoritmo è proporzionale ad N^2", lasciando non specificata la costante di proporzionalità.

Esempio: Se T(N) = $2N^2$ + 5 essendo comportamenti asintotici ossia all'infinito risultano irrilevanti ai fini della determinazione dell'ordine di grandezza sia la costante 5 sia la costante moltiplicativa 2. Ciò che interessa è dunque il termine N^2 perché formalmente

$$\lim_{N \to \infty} \frac{T(N)}{N^2} = \text{costante}$$

Quindi programmi ed algoritmi saranno valutati confrontando la loro funzione T(N) e tralasciando le loro costanti di proporzionalità. Pertanto:
- algoritmi con funzioni di complessità N, 2N, 3N oppure 2N +3, 2N + 200, 3N + 4000 sono tutti **O(N)** ossia sono proporzionali al funzione **g(N) = N**;
- algoritmi con funzioni di complessità $5N^2$, $8N^2$+ 67 sono tutti **O(N^2)** ossia sono proporzionali al funzione **g(N) = N^2**

E' possibile individuare alcuni ordini di grandezza per le funzioni **T(N)** che individuano le cosiddette **classi di complessità o di commutabilità** degli algoritmi:

1) **complessità costante O(1) oppure O(C):** indica la complessità degli algoritmi che eseguono sempre lo stesso numero di operazioni indipendentemente dalla dimensione dei dati di input (ad esempio algoritmi senza cicli);
2) **complessità logaritmica O(logN):** indica la complessità degli algoritmi che eseguono un numero di operazioni proporzionale a **logN** (ad esempio algoritmo di ricerca binaria);
3) **complessità lineare O(N):** indica la complessità degli algoritmi che eseguono un numero di operazioni proporzionale a **N** (ad esempio algoritmo di ricerca sequenziale, algoritmo di caricamento degli elementi di un vettore, algoritmo di stampa degli elementi di un vettore);
4) **complessità N logN O(N logN):** indica la complessità della maggior parte degli algoritmi di ordinamento esistenti (ad esempio l'algoritmo di MergeSort ha in tutti i casi ordine di complessità proporzionale ad N logN);
5) **complessità polinomiale $O(N^K)$:** indica la complessità degli algoritmi che hanno complessità pari alla potenza K-sima della dimensione del problema. Se K = 2 si parla di complessità **quadratica** (ad esempio l'algoritmo di ordinamento BubbleSort), se K = 3 di complessità **cubica** (ad esempio l'algoritmo di moltiplicazione tra due matrici quadrate di dimensione N);
6) **complessità esponenziale $O(K^N)$:** indica la complessità degli algoritmi che hanno complessità pari alla potenza che ha come esponente la dimensione del problema (ad esempio l'algoritmo che produce tutte le possibili stringhe di lunghezza P su di un alfabeto di 10 simboli).

Le classi di complessità dal punto di vista matematico sono **classi di equivalenza** generate dalla relazione di equivalenza **dell'ordine di grandezza**. Quindi tutti gli algoritmi che hanno lo stesso ordine di grandezza T(N) ossia hanno la stessa classe di complessità appartengono alla stessa classe di equivalenza.

EFFICIENZA DI UN ALGORITMO

Due algoritmi appartenenti alla stessa **classe di complessità** (ossia due algoritmi a parità di complessità computazionale) possono essere confrontati relativamente *al tempo di esecuzione*.
Ossia di ciascuno di essi può essere valutata l'**efficienza**.

Si dice che due algoritmi **A1** ed **A2** con funzioni di complessità rispettivamente pari ad $\mathbf{f_1(N)}$ ed $\mathbf{f_2(N)}$ che risolvono lo stesso problema **appartengono alla stessa classe di complessità** se

$$\lim_{N \to \infty} \frac{\mathbf{f_1(N)}}{\mathbf{f_2(N)}} = \mathbf{C} \quad \textbf{con C > 0 costante}$$

In particolare diremo che:
- A1 è più efficiente di A2 se:

$$\lim_{N \to \infty} \frac{\mathbf{f_1(N)}}{\mathbf{f_2(N)}} = \mathbf{C} \quad \textbf{con C < 1}$$

- A2 è più efficiente di A1 se:

$$\lim_{N \to \infty} \frac{\mathbf{f_1(N)}}{\mathbf{f_2(N)}} = \mathbf{C} \quad \textbf{con C > 1}$$

Osservazione importante
La complessità computazionale di un algoritmo, come abbiamo visto, è definita come l'ordine di grandezza della funzione che determina il numero di operazioni da svolgere per eseguirlo al crescere della dimensione dei dati in input.
E' in un certo senso **una misura di tempo astratta** che ha una relazione solo di **proporzionalità** con il **tempo effettivo di esecuzione**.
Per sapere un dato algoritmo quanto **tempo macchina effettivo** impiegherà basterà conoscere quanto tempo macchina occorre all'elaboratore in questione per eseguire una operazione elementare e moltiplicare poi per il numero complessivo di operazioni stimate.

PROBLEMI INTRATTABILI

E' necessario guardare con molta diffidenza agli algoritmi dotati di complessità computazionale esponenziale poiché necessitano di tempi di esecuzione proibitive anche per valori non molto grandi di N.
Purtroppo esistono molti problemi, anche di interesse pratico, per i quali non si conoscono algoritmi non esponenziali. Chiameremo **intrattabili** tali problemi.

A volte è una fortuna che un problema sia intrattabile; basti pensare ai sistemi di crittografia che garantiscono una elevata sicurezza perché gli algoritmi di decrittografia (che cercano di decifrare un messaggio senza conoscere la parola chiave) sono di complessità esponenziale.

CONSIDERAZIONI FINALI

Ciò che è stato detto finora permette di affrontare un qualsiasi algoritmo sotto una nuova ottica, diversa da quella precedente che possiamo riassumere come l'ottenimento dei risultati voluti.
Questi infatti potrebbero arrivare subito o anche non arrivare mai o addirittura arrivare dopo secoli di tempo di elaborazione per dimensione del problema maggiore.

Quindi è indispensabile per un buon analista alla ricerca di un algoritmo che risolva un problema valutare attentamente l'ordine di complessità di quello trovato non accontentandosi del primo trovato soprattutto se tale ordine di complessità risultasse troppo elevato o addirittura esponenziale (intrattabile).