

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.

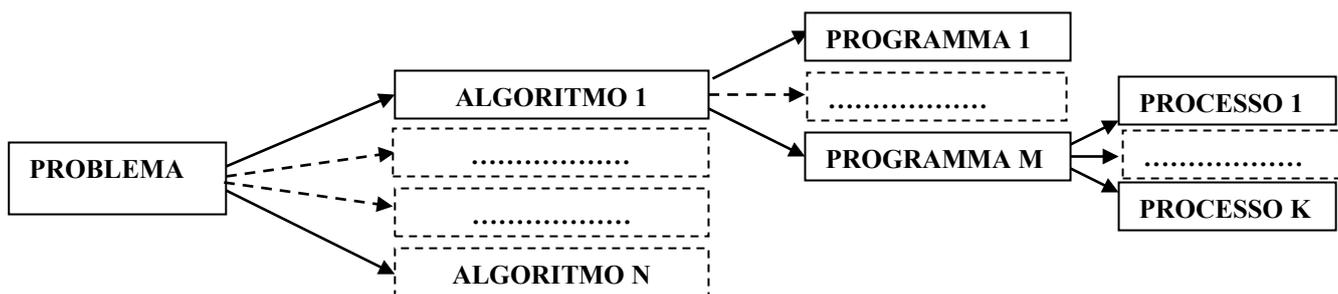
Quindi in informatica ci si occupa spesso durante la fase di analisi, per quanto possibile, di trovare “buone soluzioni” di un problema ossia ci si sforza di trovare soluzioni quanto più “economiche” possibile in termini di “efficienza” in grado di risparmiare per quanto possibile “tempo di esecuzione” 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 obiettivo 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 EQUIVALENTI quello migliore confrontandoli dal punto di vista dell’efficienza sulla base di un’analisi qualitativa delle loro 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 cronometrare 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

Introduciamo le seguenti **regole di valutazione** per potere effettuare il **calcolo del costo** di un algoritmo dettagliato in un linguaggio di progettazione strutturato (esempio **PSEUDOCODIFICA**):

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,... si ipotizza abbiano SEMPRE un costo pari ad 1;

3) Le **istruzioni iterative** (quali **MENTRE...FINE MENTRE**, **RIPETI..FINCHE'**, **PER...FINE PER**) hanno TUTTE un costo pari alla somma dei costi del test (ossia 1) e del costo delle istruzioni contenute nel corpo del ciclo, moltiplicato il numero di volte per il quale esso viene eseguito.

Si ribadisce che:

- il *test del ciclo*, viene SEMPRE considerata un'istruzione semplice e quindi si ipotizza abbia costo uguale ad 1;

- il *corpo del ciclo* avrà costo pari alla somma dei costi delle singole istruzioni calcolati in accordo con le regole qui proposte.

N.B. per i cicli post-condizionali va tenuta in considerazione una verifica di test aggiuntiva.

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 calcolate in accordo alle regole qui proposte.

L'**istruzione di selezione binaria** **SE...ALLORA ...ALTRIMENTI** 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 calcolato in accordo alle regole qui proposte.

L'**istruzione di selezione enaria** **NEL CASO CHE FINE CASO** 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 calcolato in accordo alle regole qui proposte.

5) L'**istruzione di chiamata di un sottoprogramma** (funzione o procedura) ha costo pari alla somma del costo dell'istruzione di chiamata (che per convenzione è pari ad 1) più il costo dell'intero sottoprogramma calcolato in accordo alle regole qui proposte;

N.B L'istruzione RITORNA si ipotizza abbia costo 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 calcolati in accordo alle regole qui proposte.

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 finalmente dire che, utilizzando questa tecnica di valutazione, **il tempo di esecuzione di un qualsiasi algoritmo è proporzionale al suo costo.**

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 una stima attendibile riguardo al **tempo totale di esecuzione** dell'algoritmo.

Esempio 1: calcolare il costo del seguente algoritmo:

ALGORITMO Alg_A

i, n, som: INT

INIZIO

Scrivi(“Inserisci un numero”)

Leggi (N)

$i \leftarrow 1$

som $\leftarrow 0$

MENTRE ($i \leq N$) **ESEGUI**

 som \leftarrow somma (i, som)

$i \leftarrow i + 1$

FINE MENTRE

Scrivi (“La somma richiesta è”)

Scrivi (som)

FINE

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

s: INT

INIZIO

$s \leftarrow x + y$

RITORNA (s)

FINE

Supponendo che sia $N = 10$ allora

costo Alg_A = 4 + (1 + 1 + 1 + costo chiamata somma() + costo somma()) * 10 + 1 + 2

costo Alg_A = 4 + (3 + 1 + 2) * 10 + 1 + 2 = **67**

N.B. Il costo della funzione somma() è pari a 2, poiché l’istruzione **RITORNA** ha costo pari ad 1

in generale per un **N generico**

costo Alg_A = 4 + (1 + 1 + 1 + costo chiamata somma() + costo somma()) * N + 1 + 2

costo Alg_A = **6N + 7**

N.B. Si osservi che dalla relazione **6N + 7** per $N = 10$ si ottiene esattamente il valore **67** calcolato prima

Esempio 2: calcolare il costo del seguente algoritmo:

ALGORITMO Alg_B

x, y, r, s, z :INT

INIZIO

Leggi(x)

Leggi(y)

$s \leftarrow 0$

SE ($x < 0$)

ALLORA

SE ($y < 0$)

ALLORA

$r \leftarrow 1$

ALTRIMENTI

$r \leftarrow 2$

$s \leftarrow 1$

FINE SE

$z \leftarrow 1$

FINE SE

FINE

costo Alg_B = 3 + 1 + costo del SE esterno (ALLORA)

costo del SE esterno (ALLORA) = 1 + costo del SE interno (ALTRIMENTI) + 1 = 1 + 2 + 1 = 4

costo Alg_B = 3 + 1 + 4 = **8**

Esempio 3: calcolare il costo dei seguenti algoritmi:

ALGORITMO Alg_C1

i: INT
INIZIO
 i ← 1
MENTRE (i ≤ 10) **ESEGUI**
 Scrivi("Ciao!")
 i ← i + 1
FINE MENTRE
FINE

$$\text{Costo Alg_C1} = 1 + \overset{\text{Ass}}{(10)} * \overset{\text{Ncicli}}{(1 + 1 + 1)} + \overset{\text{Test}}{1} + \overset{\text{Scrivi}}{1} + \overset{\text{Ass}}{1} + \overset{\text{Ult. test}}{1} = 1 + 31 = 32$$

Generalizzando:

$$\text{Costo Alg_C1} = 1 + \overset{\text{Ass}}{(N)} * \overset{\text{Ncicli}}{(1 + 1 + 1)} + \overset{\text{Test}}{1} + \overset{\text{Scrivi}}{1} + \overset{\text{Ass}}{1} + \overset{\text{Ult. test}}{1} = 3N + 2$$

ALGORITMO Alg_C2

i: INT
INIZIO
PER i ← 1 **A** 10 **ESEGUI**
 Scrivi("Ciao!")
 i ← i + 1
FINE PER
FINE

$$\text{Costo Alg_C2} = 1 + \overset{\text{Ass}}{(10)} * \overset{\text{Ncicli}}{(1 + 1 + 1)} + \overset{\text{Test}}{1} + \overset{\text{Scrivi}}{1} + \overset{\text{Ass}}{1} + \overset{\text{Ult. test}}{1} = 1 + 31 = 32$$

Generalizzando:

$$\text{Costo Alg_C2} = 1 + \overset{\text{Ass}}{(N)} * \overset{\text{Ncicli}}{(1 + 1 + 1)} + \overset{\text{Test}}{1} + \overset{\text{Scrivi}}{1} + \overset{\text{Ass}}{1} + \overset{\text{Ult. test}}{1} = 3N + 2$$

ALGORITMO Alg_C3

i: INT
INIZIO
 i ← 1
RIPETI
 Scrivi("Ciao!")
 i ← i + 1
FINCHE' (i > 10)
FINE

$$\text{Costo Alg_C3} = 1 + \overset{\text{Ass}}{(10)} * \overset{\text{Ncicli}}{(1 + 1 + 1)} + \overset{\text{Test}}{1} + \overset{\text{Scrivi}}{1} + \overset{\text{Ass}}{1} = 1 + 30 = 31$$

Generalizzando:

$$\text{Costo Alg_C3} = 1 + \overset{\text{Ass}}{(N)} * \overset{\text{Ncicli}}{(1 + 1 + 1)} + \overset{\text{Test}}{1} + \overset{\text{Scrivi}}{1} + \overset{\text{Ass}}{1} = 3N + 1$$

N.B. in questo caso seppur in presenza di un impercettibile vantaggio nell'utilizzo di una ciclo di tipo RIPETI esso è di fatto trascurabile al crescere della dimensione del problema N.

Esempio 4: calcolare il costo del seguente algoritmo:

ALGORITMO Alg_D

i: INT
INIZIO
 i ← 1
MENTRE (i ≤ N) **ESEGUI**
 SE (i % 2 = 0)
 ALLORA
 k ← i * 3
 j ← j + k
 ALTRIMENTI
 k ← k / 2
 Scrivi (k)
 FINE SE
 SE (k + i % 3)
 ALLORA
 k ← 2*i
 FINE SE
 Scrivi ("Uffa!")
 i ← i + 1
FINE MENTRE
FINE

Svolgimento:

$$\text{Costo Alg_D} = 1 + \overset{\text{Ass}}{N} * \overset{\text{Ncicli}}{(1 + \overset{\text{Test M}}{\text{Costo SE Bin}} + \overset{\text{Ass}}{\text{Costo SE Un}} + 1 + 1)} + \overset{\text{Ult.test M}}{1}$$

dove

$$\text{Costo SE Bin} = \overset{\text{Test}}{1} + \overset{\text{Ass}}{1} + \overset{\text{Ass}}{1} = 3$$

$$\text{Costo SE Un} = \overset{\text{Test}}{1} + \overset{\text{Ass}}{1} = 2$$

Quindi sostituendo

$$\text{Costo Alg_D} = 1 + \overset{\text{Ass}}{N} * \overset{\text{Ncicli}}{(1 + 3 + 2 + 1 + 1)} + \overset{\text{Test M}}{1} + \overset{\text{Ass}}{1} + \overset{\text{Incr}}{1} + \overset{\text{Ult.test M}}{1} = 8N + 2$$

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.

La **funzione T(N)** è chiamata **complessità computazionale** (*in tempo*) o semplicemente **complessità** di un algoritmo.

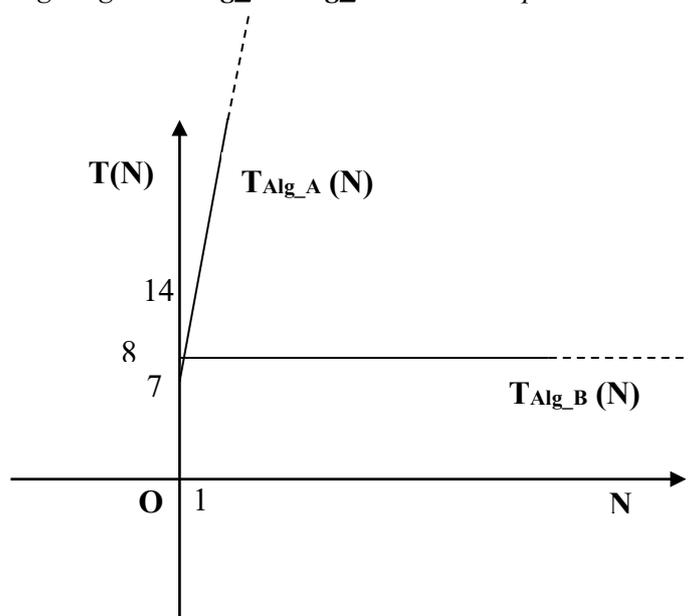
La **dimensione N** dei dati in input è anche detta **dimensione del problema**.

Essendo T(N) una funzione matematica è possibile rappresentarla attraverso il suo grafico in un sistema di assi cartesiani 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: Grafici dei costi degli algoritmi Alg_A e Alg_B calcolati in precedenza



Esercizi di approfondimento: svolgere gli esercizi contenuti nei seguenti file PDF

1) <http://www.riochierego.it/mobile/docs/quarta/lab/Esercizi-Calcolo-Complessita-Computazionale-LIBRO-SVOLTI.pdf>

2) <http://www.riochierego.it/mobile/docs/quarta/lab/Esercizi-Calcolo-Complessita-Computazionale-RIO.pdf>

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, non essendo possibile un calcolo esatto, si esegue una stima sulla base del caso **migliore** e del caso **peggiore**, indicando ove possibile anche il caso **medio**.

Quindi in generale si prevedono tre tipi di analisi ossia:

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

Quindi la funzione complessità **T(N)** va SEMPRE analizzata nel caso più favorevole, in quello sfavorevole e nel caso medio in base alla **dimensione** ed alla **disposizione** dei dati in ingresso.

Per esprimere tali funzioni di complessità utilizzeremo le seguenti notazioni:

- a) **T_{ottimo}(N)**;
- b) **T_{pessimo}(N)**;
- c) **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).

Ricerca di un elemento in un insieme ordinato di elementi

(per esempio un numero telefonico in un elenco ordinato alfabeticamente o un numero intero in un insieme ordinato di numeri interi)

FUNZIONE Ric_Seq (VAL v ARRAY[] 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 ARRAY[] 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

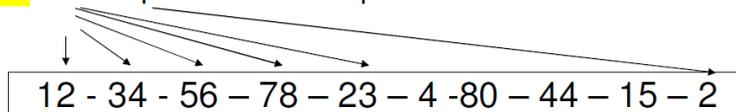
FINE MENTRE

RITORNA (trovato)

FINE

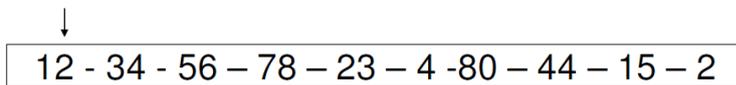
Ricerca di un elemento x: Algoritmo RICERCA SEQUENZIALE

x = 2 - 10 passi necessari per trovare il numero 2



- Caso peggiore - 10 confronti su 10 numeri

x = 12 - un passo per raggiungere il numero 12



- Caso migliore - 1 confronto su 10 numeri
- Caso medio $(1+10)/2 \rightarrow (1+N)/2 \rightarrow 5,5$ che arrotondato 5

Ipotizziamo ora che gli elementi dell'insieme siano 100

- **Caso ottimo** ossia $T_{ottimo}(N) = 1$

- **Caso pessimo** ossia $T_{pessimo}(N) = 100$

- **Caso medio** ossia $T_{medio}(N) = (se\ l'elemento\ è\ nella\ prima\ posizione\ il\ numero\ dei\ tentativi\ è\ 1) + (se\ l'elemento\ è\ nella\ seconda\ posizione\ il\ numero\ dei\ tentativi\ è\ 2) + ecc.. = (1 + 2 + 3 + ... + 100) / 100$

Più in generale, se ipotizziamo che gli elementi dell'insieme siano N, allora il caso medio (pari al numero di confronti medio da effettuare per trovare l'elemento) sarà espresso dalla relazione:

$$T_{medio}(N) = (1 + 2 + 3 + ... + N) / N$$

Poiché $(1 + 2 + 3 + ... + N) = N * (N+1)/2$ sostituendo avremo

$T_{medio}(N) = (N+1)/2$ che se N è 100 vale 50 poiché $(100+1)/2 = 50.5$ ed arrotondiamo per difetto

Ricerca di un elemento x in un insieme ORDINATO: Algoritmo RICERCA BINARIO

Si suppone di avere un insieme ordinato di valori (esempio elenco telefonico) e di volerne cercare uno al suo interno (esempio "Mario Rossi").

Si suppone di dividere l'insieme ordinato in due parti e vedere se il valore cercato cade nella prima parte dell'insieme o nella seconda. Si identifica così la parte dell'insieme ordinato su cui lavorare e si procede ricorsivamente ossia applicando lo stesso procedimento al risultato del passo precedente.

In questo modo si scarta tutta la parte di elementi che sicuramente sono inutili ai nostri fini e ci si concentra sulla parte utile

Si riduce il problema della metà (circa) a ogni interazione limitando la ricerca alla parte precedente o alla parte successiva del valore che si trova a metà dell'insieme (nel suo *centro*).

Ogni tentativo permette di **ridurre** l'ampiezza del problema **di un fattore 2** (perciò detto "binario").

Il primo tentativo riduce il numero degli elementi da confrontare a $N/2$; il secondo a $N/4$, il terzo a $N/8$, e così via. Quindi:

- **Caso pessimo**: il numero di passi (confronti) da effettuare è il numero **K** equivalente al numero di divisioni che consente di arrivare all'ultimo elemento dell'insieme ossia continuo a dividere **N** finché quello che rimane è un solo elemento ossia

$$N/2/2/2/.../2 = 1 \quad \text{allora ho} \quad N/2^k = 1 \quad \text{ossia } N = 2^k \quad \text{ossia } K = \log_2 N \quad \text{quindi}$$

$$T_{\text{pessimo}}(N) = \log_2 N$$

- **Caso ottimo** ossia $T_{\text{ottimo}}(N) = 1$ quando l'elemento da ricercare si trova esattamente nella posizione $N/2$

- **Caso medio** ossia $T_{\text{medio}}(N) = (1 + \log_2 N)/2$ che appartiene comunque alla famiglia computazionale $\log_2 N$

Tabella riassuntiva comparativa dei due algoritmi di RICERCA

	Ricerca sequenziale	Ricerca binaria
Caso ottimo	1	1
Caso pessimo	N	$\log_2 N$
Caso medio	$N / 2$	$1 + \log_2 N$

N.B. Come è possibile osservare gli algoritmi hanno la stessa complessità nel caso ottimo, ma complessità diverse nel caso medio e nel caso pessimo.

In caso di ricerca di un elemento in un insieme ordinato, la maggiore efficienza dell'algoritmo di ricerca binaria o dicotomica rispetto a quello di ricerca sequenziale è plasticamente rappresentato dai dati presenti in questa tabella che specificano i numeri di confronti effettuati dai due algoritmi NEL CASO MEDIO al crescere della dimensione **N** dei dati del problema

Misure nel caso medio

Valore di <i>N</i>	Ricerca lineare <i>N/2</i>	Ricerca binaria <i>Log₂ N</i>
10	5	~4
100	50	~7
1 000	500	~10
10 000	5.000	~ 14
100 000	50.000	~ 17
1 000 000	500.000	~ 20

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

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

$$\lim_{N \rightarrow \infty} T(N)$$

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

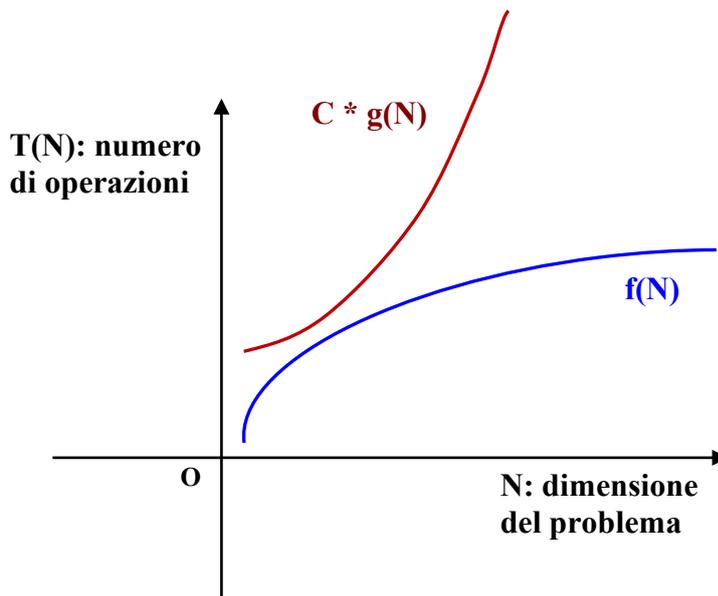
Introduciamo le seguenti definizioni:

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 $f(N)$ è $O(g(N))$ significa dire che $g(N)$ è un limite superiore **alla crescita di $f(N)$** ossia che $f(N)$ 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

$$f(N) \text{ è } O(g(N)) \text{ e } g(N) \text{ è } O(f(N))$$

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 C 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

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

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 $5N^2$, $8N^2 + 67$, $19N^2 + 67N + 2$ sono tutti $O(N^2)$ ossia sono proporzionali al funzione $g(N) = N^2$

CLASSI DI COMPUTABILITA'

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

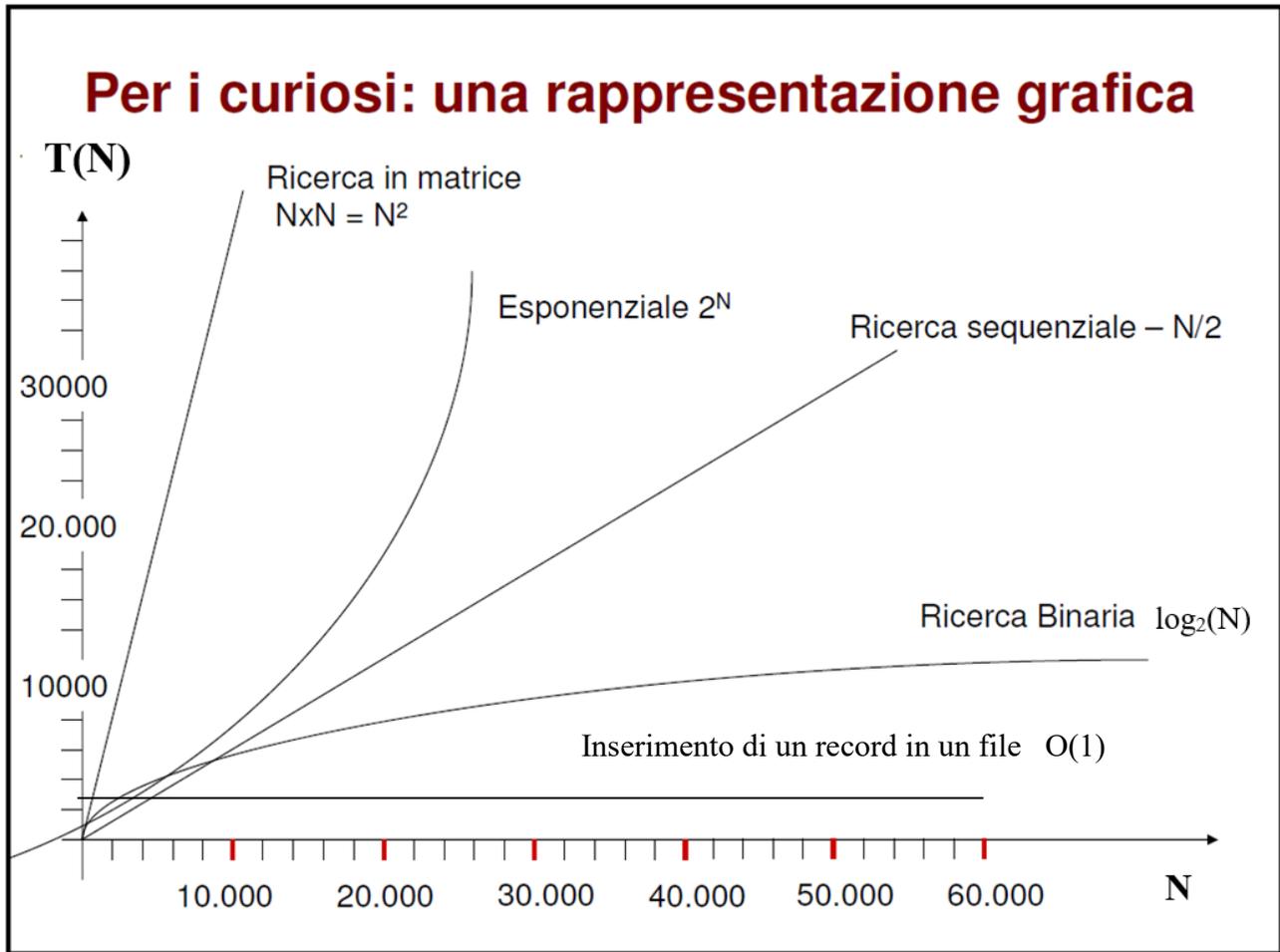
- 1) **complessità COSTANTE $O(1)$ oppure $O(C)$** : indica la complessità degli algoritmi che eseguono sempre lo stesso numero di operazioni indipendentemente dalla dimensione dei dati di input (ad esempio algoritmi senza cicli);
- 2) **complessità LOGARITMICA $O(\log N)$** : indica la complessità degli algoritmi che eseguono un numero di operazioni proporzionale a $\log N$ (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 PSEUDOLINEARE $O(N \log N)$** : 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 $N \log N$);
- 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).

I problemi con complessità al più polinomiale si definiscono TRATTABILI;

I problemi con complessità almeno esponenziale si definiscono INTRATTABILI

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.



Classi di complessità

n	$\log_2 n$	$n \cdot \log_2 n$	n^2	n^3	2^n
2	1	2	4	8	4
10	3,322	33,22	10^2	10^3	$> 10^3$
10^2	6,644	664,4	10^4	10^6	$\gg 10^{25}$
10^3	9,966	9996,0	10^6	10^9	$\gg 10^{250}$
10^4	13,287	1328,7	10^8	10^{12}	$\gg 10^{2500}$

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 $f_1(N)$ ed $f_2(N)$ che risolvono lo stesso problema **appartengono alla stessa classe di complessità** se

$$\lim_{N \rightarrow \infty} \frac{f_1(N)}{f_2(N)} = C \quad \text{con } C > 0 \text{ costante}$$

In particolare diremo che:

- **A1 è più efficiente di A2** se:

$$\lim_{N \rightarrow \infty} \frac{f_1(N)}{f_2(N)} = C \quad \text{con } C < 1$$

- **A2 è più efficiente di A1** se:

$$\lim_{N \rightarrow \infty} \frac{f_1(N)}{f_2(N)} = C \quad \text{con } C > 1$$

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.

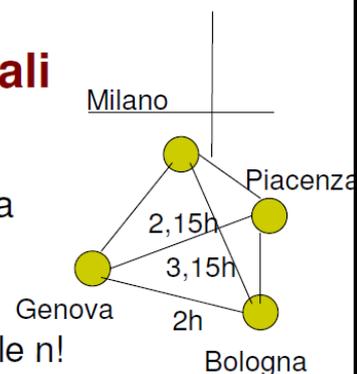
E' in un certo senso **una misura di tempo astratta** che ha una relazione solo di **proporzionalità** con il **tempo effettivo di esecuzione**.

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.

Esempi di problemi esponenziali

- Problema del **commesso viaggiatore**

In una rete di X città visitarle solo una volta con percorso minimo



- Problema con complessità computazionale $n!$
- Rompere una chiave di crittografia
- Attacco informatico di forza bruta
- Problemi di connessione e di instradamento in Internet (routing)
- Combinazione delle intercettazioni telefoniche
- Costruzione di circuiti
- Logistica dei trasporti
- Google map

Algoritmi euristici abbassano la complessità computazionale da esponenziale a polinomiale

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).