Analisi di Complessità

Prof. Sebastiano Battiato

Prof.ssa Rosalba Giugno

Complessità Computazionale

Uno stesso problema può spesso essere risolto da algoritmi di diversa efficienza.

Le differenze tra algoritmi possono essere di poco conto nella manipolazione di un piccolo numero di dati ma crescono proporzionalmente all'aumentare della dimensione dell'input.

La Complessità Computazionale indica quale sforzo sia necessario per applicare un algoritmo o quanto questo sia costoso: tale costo può essere misurato in diversi modi ed il contesto in cui lo si usa ne determina il significato.

Problema

Un **problema** è una funzione fra 2 insiemi: il primo è l'insieme delle istanze del problema, il secondo è l'insieme delle soluzioni.

Esempio 1. Prodotto aritmetico.

$P : \text{Istanze}_P \rightarrow \text{Soluzioni}_P$

$(2,3)\ \alpha\ 6$

$(2,7)\ \alpha\ 14$

$(3,4)\ \alpha\ \ldots$

Esempio 2. Ordinamento non decrescente di n numeri naturali.

Algoritmo

Un algoritmo è un procedimento finito, non ambiguo, terminante.

Finito: descrivibile finitamente

Non ambiguo: ad ogni istante e' determinabile con probabilita' unitaria cio' che l'algoritmo sta eseguendo

Terminante: avente sempre fine

Molteplici esempi.

Un ciclo infinito non è un algoritmo secondo la definizione di cui sopra.

Nel mondo reale.....

Gli algoritmi intervengono (in modo più o meno nascosto) in molti aspetti dell'informatica

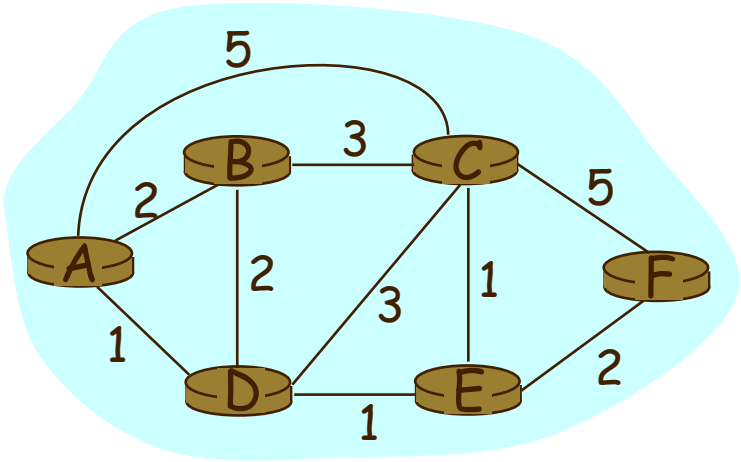
Esempi

- Routing Internet
- DBMS
- Analisi e classificazione di documenti/Motori di ricerca
- Ottimizzazione di progetti
- ecc. ecc.

Esempio: Routing in Internet

Protocollo di Routing

Obiettivo: determinare "buon" cammino sorgente-destinazione



Astrazione usando strutture dati ad-hoc dette "grafi":

- I nodi rappresentano router
- Gli archi descrivono i link fisici
  - Costi sugli archi (link) : ritardo, costo in €, livello di congestione

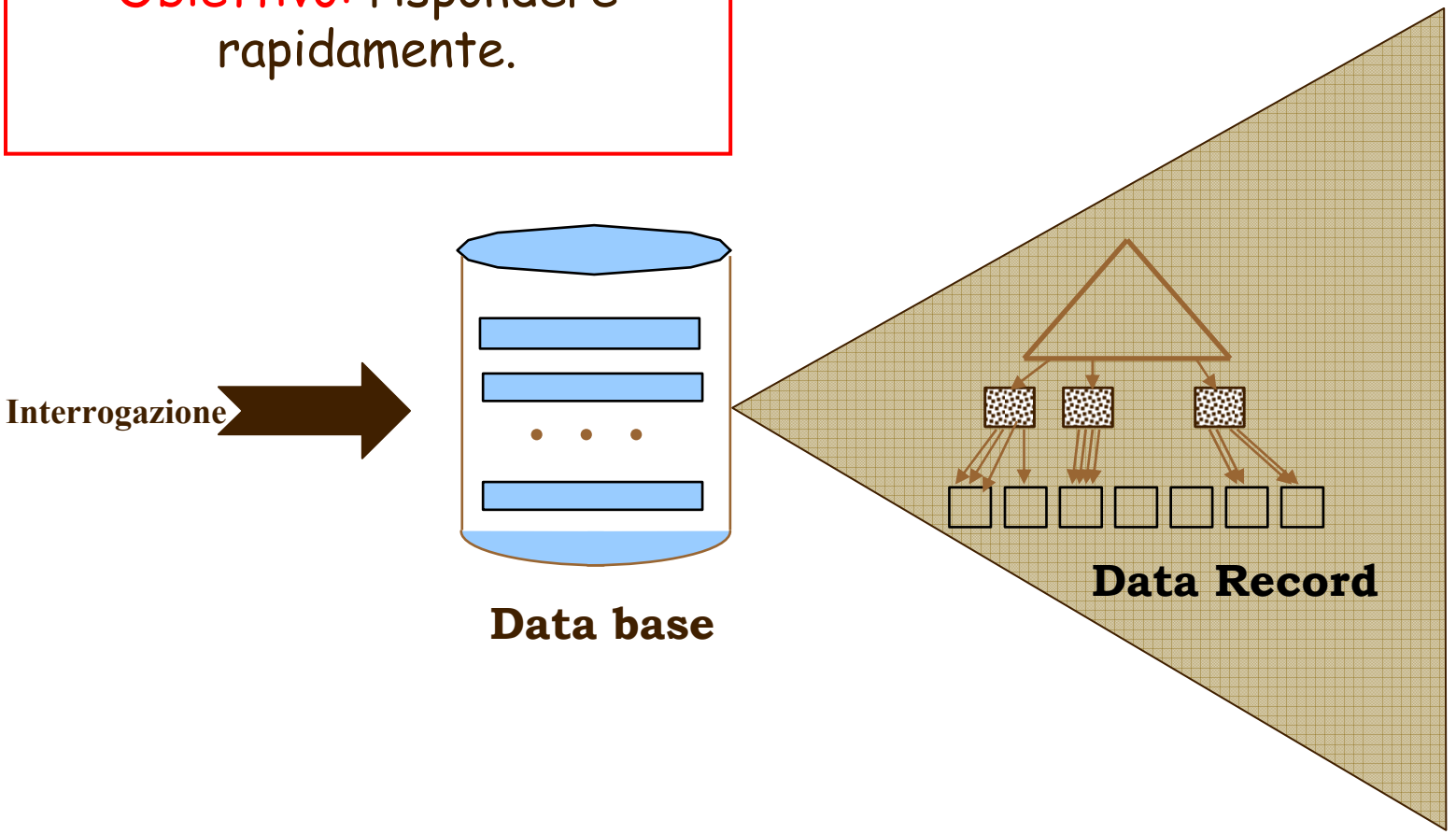
Cammino "buono":

- Di solito significa cammino a costo minimo
- Possibili def. alternative

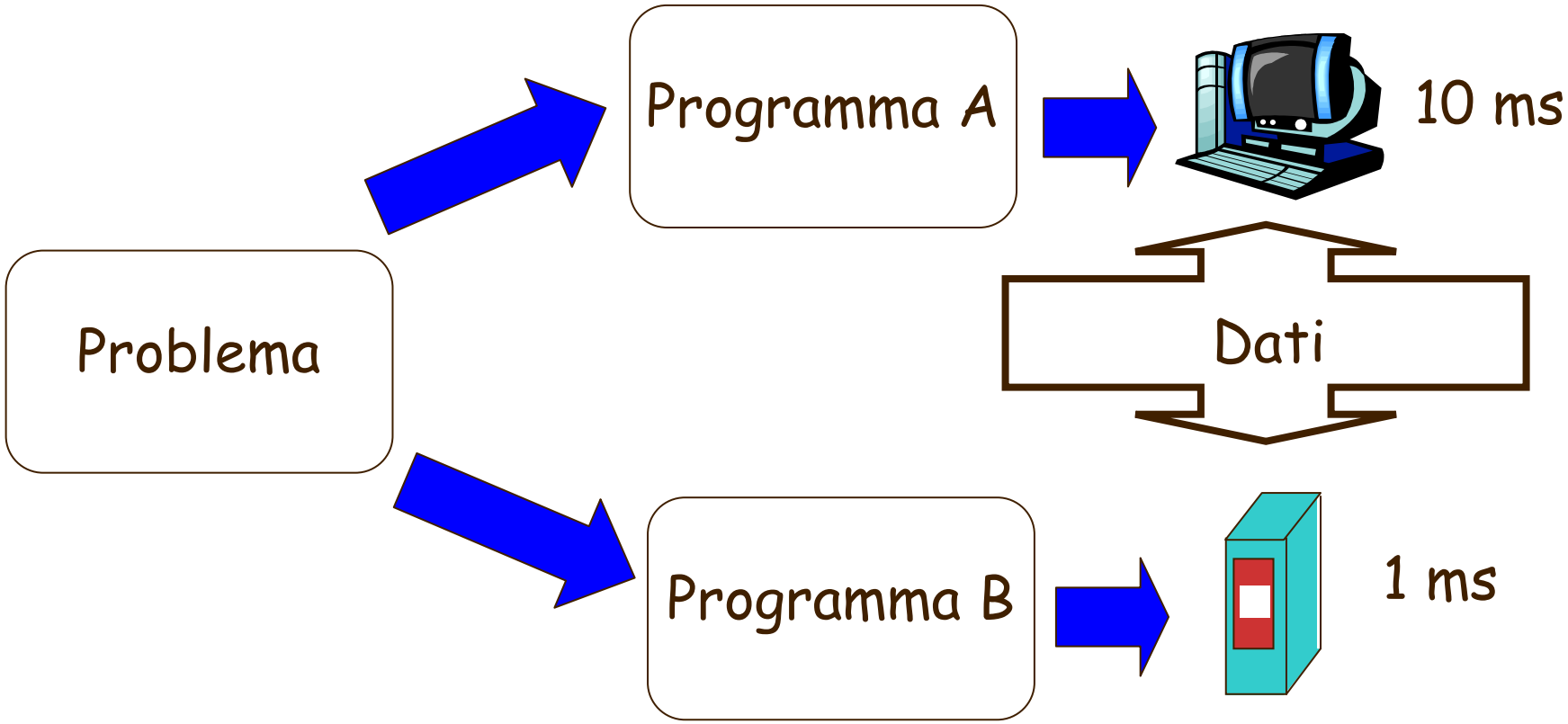
Esempio: Accesso a Basi di Dati

Interrogazione

Obiettivo: rispondere rapidamente.



Come Misurare l'Efficienza?



Perché B gira più velocemente ?

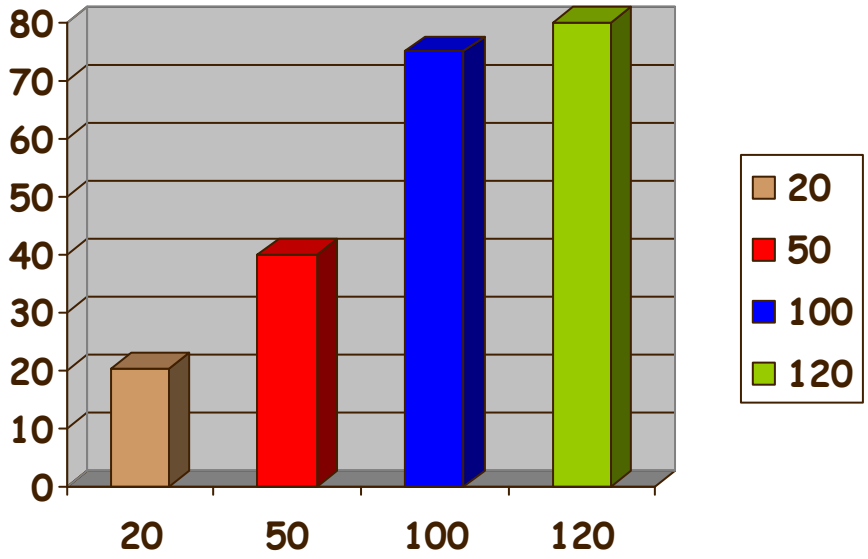
Possiamo affermare che B sia "migliore" di A?

Tempo di Esecuzione

In generale aumenta con la dimensione dell'Input.

Per lo stesso valore di *n* può variare con il particolare input considerato (es. Vettore da ordinare)

Dipende dalla piattaforma Hw/Sw



Può essere poco rappresentativo!

Analisi di Complessità

Non è conveniente valutare la complessità di un algoritmo dal ***tempo*** di esecuzione di un programma che lo realizzi, perché dipenderebbe

- dalle modalità di implementazione,
- dallo specifico input,
- dall'hardware utilizzato

Ogni analisi di complessità richiederebbe l'implementazione e l'esecuzione dell'algoritmo

Analisi di Complessità

Esistono molteplici criteri, basati ad esempio su

- Tempo di comunicazione tra le varie periferiche
- Tempo di comprensione dell'algoritmo per il programmatore medio
- Tempo di implementazione dell'algoritmo per il programmatore medio (talvolta in anni-uomo)
- Spazio di memoria primaria occupato dall'implementazione (di cui al punto precedente) in esecuzione
- Spazio di memoria secondaria occupato per memorizzare permanentemente l'implementazione
- …

Verso un'Analisi Teorica

Vorremmo una stima della complessità che prescinda dalle caratteristiche dell'implementazione, dell'input, e dell'hardware.

- i compilatori esistenti richiedono più o meno lo stesso numero di istruzioni macchina per tradurre un'istruzione sorgente
- tutti i comuni elaboratori sono basati sull'architettura Von Neumann

Verso un'Analisi Teorica

Quindi, sia l'implementazione, sia l'hardware incidono al più per un fattore costante sul tempo di esecuzione complessivo di un algoritmo.

Esempio. Consideriamo un'implementazione per un algoritmo, che impieghi tempo $T(n)$ su una macchina M.

Allora un'altra implementazione per lo stesso algoritmo impiegherà tempo $c \cdot T(n)$ su una qualunque altra macchina M'.

Verso un'Analisi Teorica

Inoltre, il tempo di esecuzione complessivo di un algoritmo dipende dalla **dimensione** dell'input, ossia dalla quantità di memoria necessaria a rappresentare l'input.

Esempio. Scegliamo un algoritmo per ordinare una lista di numeri. Supponiamo che l'algoritmo impieghi tempo t per ordinare 5, 3, 1, 4 .

Allora esso impiegherà un tempo diverso da t per ordinare una sequenza di lunghezza 1000.

Funzioni di Complessità

In un'analisi teorica, è sufficiente osservare che il tempo di esecuzione di un algoritmo dipende dalla dimensione dell'input.

La **funzione di complessità tempo** di un algoritmo lega la dimensione dell'input al tempo che l'algoritmo impiega su di esso. Esprime l'evoluzione del tempo di computazione al variare della dimensione dell'input.

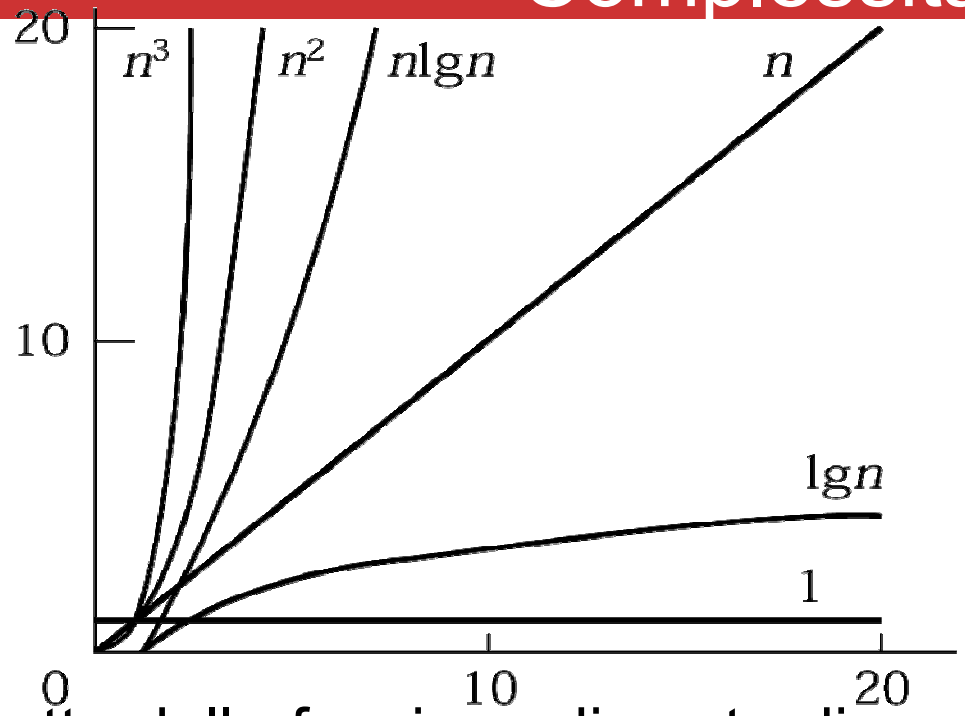
Funzioni di Complessità

Dato un algoritmo A, tipicamente T_A indica la sua funzione di complessità.

Se non ci sono ambiguità, la funzione di complessità si indica semplicemente con T.

Analogamente, si può definire la funzione di **complessità spazio** S_A.

Complessità Asintotica



Il calcolo esatto della funzione di costo di un algoritmo è nella maggioranza dei casi molto difficile e può essere semplificato senza perdere di significatività:

- Trascurando i fattori costanti

- Descrivendo l'andamento di **T** quando ***n*** tende all'infinito.

Analisi Asintotica

Se si considerano tutte le costanti l'analisi può divenire eccessivamente complessa. Interessa conoscere il costo al variare della dimensione dell'input, a meno di costanti.

Un'analisi di questo tipo è detta ***asintotica***

L'analisi asintotica è influenzata dall'*operazione dominante* di un algoritmo

Approssimazioni

L'**analisi asintotica** di complessità studia il comportamento asintotico della funzione di costo di un metodo/algoritmo. L'idea è considerare solo il comportamento della funzione di costo al crescere della dimensione dei dati di ingresso (tipicamente, ci interessano valori di n molto grandi)

L'analisi asintotica è una tecnica di analisi **approssimata:**

– l'idea è di trascurare fattori costanti e termini di ordine inferiore

– i risultati sono approssimati e non precisi

I risultati ottenuti sono comunque significativi nella maggioranza dei casi essendo indipendenti dal calcolatore usato.

Esempi di Funzioni di Complessità

Esempio

Sia c una costante positiva e $T_A(n) = c \cdot n$ e $T_B(n) = \log 2\ n$

Consideriamo due input di dimensione n_1 e $2n_1$.

Allora: $T_A(n_1) = c \cdot n_1$

$$T_A(2n_1) = c \cdot 2 \cdot n_1 = 2 \cdot T_A(n_1)$$

$$T_B(n_1) = \log_2 n_1$$

$$T_B(2n_1) = \log_2 (2 \cdot n_1) = 1 + T_B(n_1)$$

Quale degli algoritimi A e B trovate preferibile?

Approssimazioni

Esempio. $T(n) = n^2 + 100n + \log_{10} n + 1000$

n	$f(n)$	n^2		$100n$		$\log_{10} n$		1000	
	Valore	Valore	%	Valore	%	Valore	%	Valore	%
1	1101	1	0.1	100	9.10	0	0.0	1000	0.82
10	2101	100	4.76	1000	47.60	1	0.05	1000	7.62
100	21 002	10 000	47.6	10 000	47.60	2	0.991	1000	4.76
1000	1 101 003	1 000 000	90.8	100 000	9.10	3	0.0003	1000	0.09
10 000	101 001 004	100 000 000	99.0	1 000 000	0.99	4	0.0	1000	0.001
100 000	10 010 001 005	10 000 000 000	99.9	10 000 000	0.099	5	0.0	1000	0.00

- n piccolo: prevale l'ultimo termine
- n = 10: secondo e ultimo termine uguali
- n = 100: primo e secondo termine uguali
- n > 100: è il primo termine a prevalere

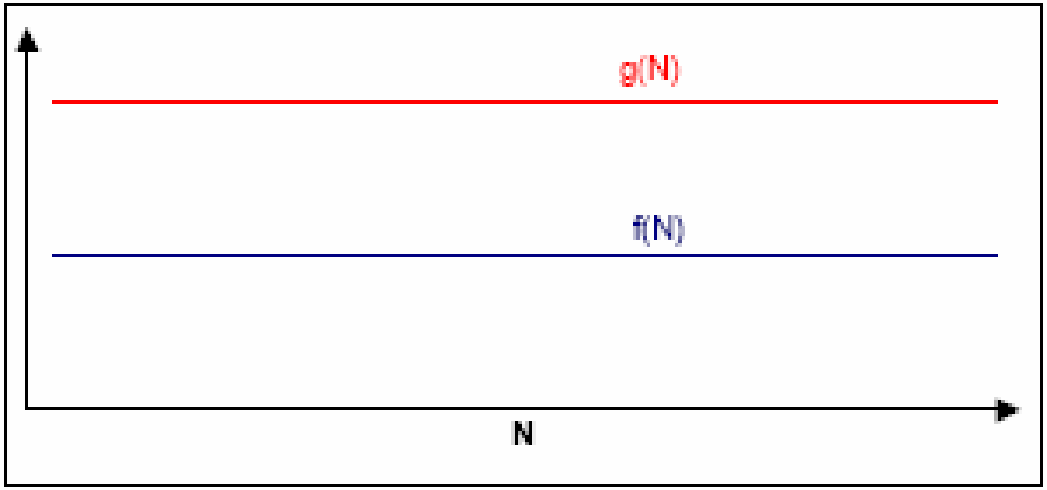
Notazione O-grande

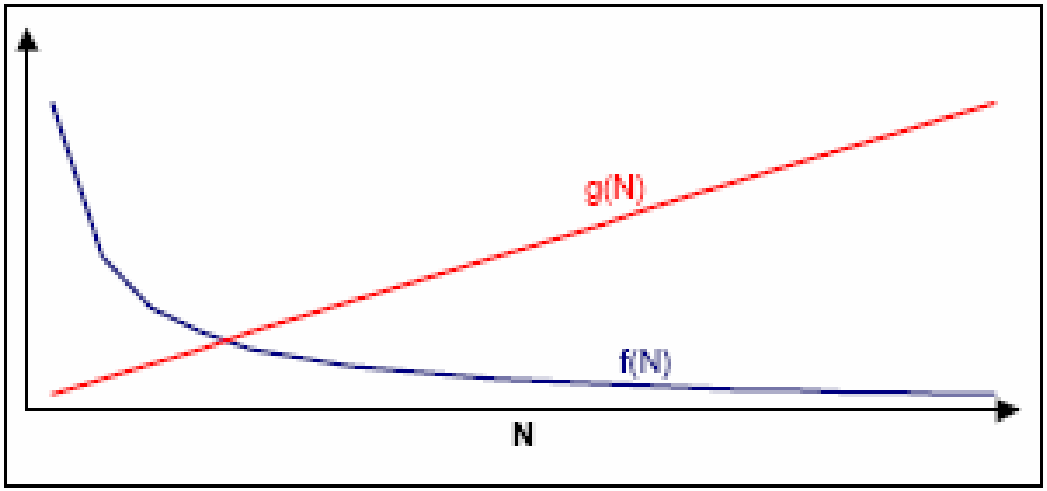
Definizione1. T(n) è O(g(n)) se esistono due numeri positivi c ed N tali che $T(n) \leq c\, g(n)$ per qualsiasi $n \geq N$.

Si scrive T(n) = O(g(n))

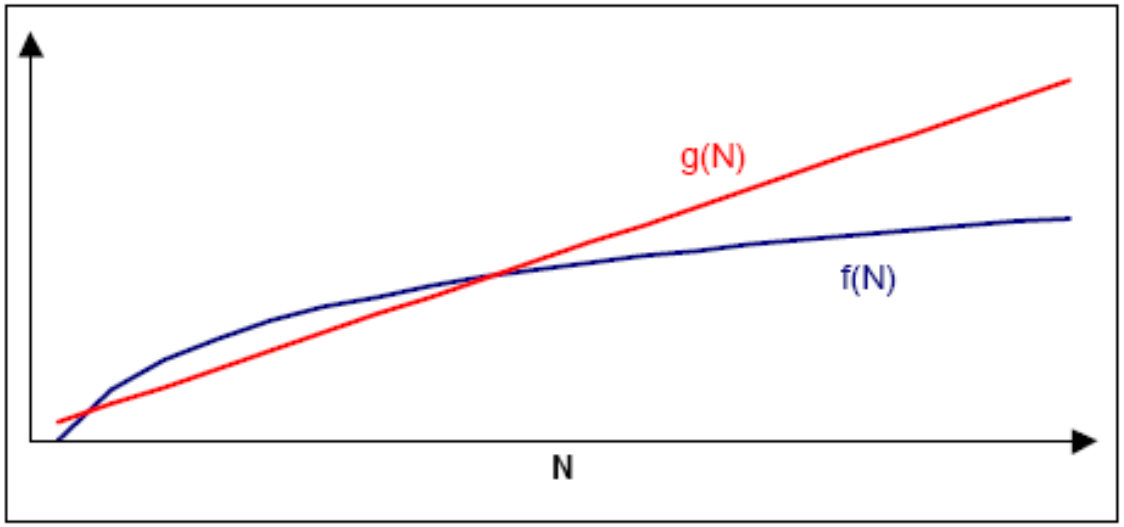
- g(n) limita superiormente T(n)
- g(n) approssima asintoticamente T(n) dall'alto
- g(n) è una buona approssimazione superiore per T(n) quando n è molto grande

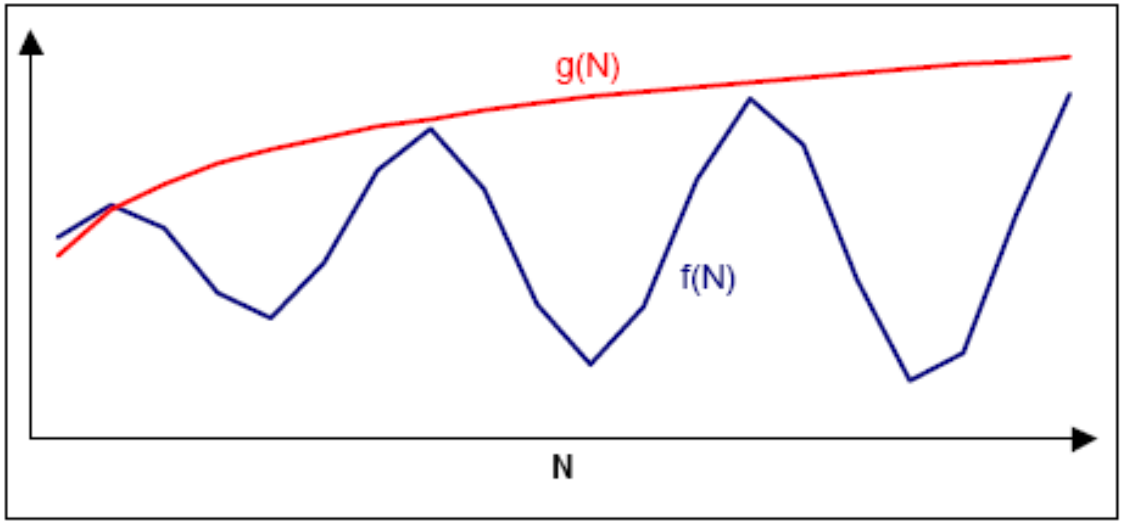
Notazione O-grande





Notazione O-grande





Notazione O-grande

Esempio. $T(n) = n^2 + 100n + \log_{10} n + 1000$

Trovare $g(n)$ tale che $T(n) = O(g(n))$.

Non immediato perché la definizione non è costruttiva.

Ma… quale dei quattro termini cresce più velocemente?

Notazioni Ω

Definizione2. T(n) è Ω(g(n)) se esistono due numeri positivi c ed N tali che $T(n) \geq c\, g(n)$ per qualsiasi $n \geq N$.

Si scrive $T(n) = \Omega(g(n))$

- g(n) limita inferiormente T(n)
- g(n) approssima asintoticamente T(n) dal basso
- g(n) è una buona approssimazione inferiore per T(n) quando n è molto grande

Notazione Θ

Definizione3. T(n) e' $\Theta(g(n))$ se esistono numeri positivi c_1, c_2, N tali che $c_1\, g(n) \leq T(n) \leq c_2\, g(n)$ per qualsiasi $n \geq N$

Si scrive $T(n) = \Theta(g(n))$

- g(n) limita T(n)
- g(n) approssima asintoticamente T(n)
- g(n) è una buona approssimazione per T(n) quando n è molto grande

Osservazioni

La caratterizzazione che ci interessa di più è quella più precisa, cioè più stringente.

Però, se ad esempio $T(n) = O(n)$

non è sbagliato dire che $T(n) = O(n \log n)$

oppure che $T(n) = O(n^2)$

Analogamente, se $T(n) = \Omega(n^2)$

non è sbagliato dire che $T(n) = \Omega(n \log n)$

oppure che $T(n) = \Omega(n)$

Proprietà

Teorema1.

$T(n) = \Theta(g(n))$ se e solo se $T(n) = O(g(n))$ e $T(n) = \Omega(g(n))$

Dimostrazione1. banale

Proprietà di Θ

Teorema2. Un polinomio con termine di grado massimo positivo si comporta asintoticamente come il monomio con potenza massima e coefficiente unitario.

Ossia:

$$\sum_{i=0}^{d} a_i n^i = \Theta(n^d) \qquad a_d > 0$$

Dimostrazione2.

$$\lim_{n \to \infty} \sum_{i=0}^{d} \frac{a_i}{n^{d-i}} = \lim_{n \to \infty} \left(a_d + \sum_{i=0}^{d-1} \frac{a_i}{n^{d-i}} \right) = a_d$$

Proprietà di Θ

$\forall \varepsilon > 0$, $\exists N \in \mathbb{N}$ tale che $\forall n \geq N$ vale $\left| \sum_{i=0}^{d} \frac{a_i}{n^{d-i}} - a_d \right| < \varepsilon$

ossia $a_d - \varepsilon < \sum_{i=0}^{d} \frac{a_i}{n^{d-i}} < a_d + \varepsilon$

Fissiamo $\bar{\varepsilon} > 0$ tale che $a_d - \bar{\varepsilon} > 0$

Abbiamo quindi trovato due costanti positive c_1 e c_2 tali da soddisfare la definizione a meno di moltiplicare tutti i membri per n^d

Esempi

- $T(n) = n^2 + 100n + \log_{10} n + 1000 \quad = \Theta(n^2)$

  In particolare vale anche,

  - $T(n) = \Omega(n^2)$
  - $T(n) = O(n^2)$

- $T(n) = n + 10^9 \quad = O(n)$
- $T(n) = n + \log n \quad = O(n)$
- $T(n) = n + 2^n \quad = O(2^n)$

Complessità tipiche

complessità \ dim. input	10	10^3	10^6
costante - O(1)	*1 μsec*	*1 μsec*	*1 μsec*
logaritmica - O(lg n)	3 *μsec*	*10 μsec*	*20 μsec*
lineare - O(n)	10 *μsec*	*1 msec*	1 sec
n lg n - O(n lg n)	33 *μsec*	*10 msec*	20 sec
quadratica - $O(n^2)$	100 *μsec*	1 sec	10^{12} sec
cubica - $O(n^3)$	1 msec	10^9 sec	10^{18} sec
esponenziale - $O(2^n)$	10 msec	10^{301} sec	10^{301030} sec

11.6 gg

31709 anni !!

16.7 min

Altre proprietà

Teorema3.

Se $T_1(n) = O(g(n))$ e $T_2(n) = O(g(n))$,

allora

$T_1(n) + T_2(n) = O(g(n))$.

Dimostrazione3. Si prenda una costante pari alla **somma** delle costanti derivanti dalle ipotesi.

Altre proprietà

Teorema4. (transitività)

Se T(n) = O(g(n)) e g(n) = O(h(n)),

allora

T(n) = O(h(n)).

Dimostrazione4. Si prenda una costante pari al **prodotto** delle costanti derivanti dalle ipotesi.

Altre proprietà

Teorema5.

base 2

$O(\log_b n) = O(\lg n)$

Dimostrazione5. Segue da $\log_b n = c \lg n$, inglobando la costante dentro la notazione.

Questi Teoremi valgono anche per Ω e Θ.

Attenzione alle Costanti

Alcuni ulteriori considerazioni:

Le costanti nascoste possono essere molto grandi: un algoritmo con costo $10^{50}n$ è lineare ma potrebbe essere poco pratico

Comportamento nel caso di istanze “piccole” (es. 3n contro n^2)

Il caso peggiore potrebbe verificarsi raramente

Tempo d'esecuzione

Volendo avere una stima effettiva del tempo impiegato dai propri algoritmi è possibile in JAVA utilizzare il metodo `currentTimeMillis()`, della classe `System`:

```
public static long currentTimeMillis()
```

Il valore di ritorno del metodo corrisponde al numero di millisecondi trascorsi dal momento in cui viene chiamato alla mezzanotte del 1 Gennaio 1970.

L'unità di misura del tempo di sistema è solitamente più granulare (1/10, 1/100 di millisecondo).

Alternativamente è possibile fare riferimento alla classe `Date` del package `java.util`

La classe StopWatch

```
public class StopWatch {
    private long elapsedTime;
    private long startTime;
    private boolean isRunning;

    public StopWatch(){ reset(); }

    /** Ferma il cronometro e azzera il tempo totale
        trascorso.*/
        public void reset(){
        elapsedTime = 0;
        isRunning = false;
        }

    /* Fa partire il cronometro */
    public void start(){
    if (isRunning) return;
        isRunning = true;
        startTime = System.currentTimeMillis();   }
```

La classe StopWatch

```
/* Ferma il cronometro. Il tempo non viene più accumulato
   e viene sommato al tempo trascorso.*/

    public void stop(){
        if (!isRunning) return;
        isRunning = false;
        long endTime = System.currentTimeMillis();
        elapsedTime = elapsedTime+endTime-startTime;
    }

    /* Restituisce il tempo totale trascorso.*/

    public long getElapsedTime(){
        if (isRunning)
        {
         long endTime = System.currentTimeMillis();
         elapsedTime = elapsedTime+endTime-startTime;
         startTime = endTime;
        }
        return elapsedTime;    }
```

Esempio

```
…
StopWatch timer = new StopWatch();

            timer.start();
            {…
             Sorts.insertionSort( aux );
             …
             }
            timer.stop();

System.out.println("Elapsed time Insertion Sort: "+
timer.getElapsedTime() + " milliseconds");
```

In genere, un problema può essere risolto da più algoritmi. L'analisi di complessità fornisce uno strumento per scegliere il "migliore" algoritmo tra quelli disponibili

Supponiamo ad esempio di dover risolvere un problema **P** e di conoscere gli algoritmi **A** e **B** per risolvere **P**

– supponiamo che **A** sia $\Theta(N^2)$ e che **B** sia $\Theta(N \log N)$

– possiamo concludere che **B** è asintoticamente migliore di **A**, e quindi decidere di risolvere il problema **P** implementando l'algoritmo **B**

Complessità e Algoritmi:un esempio

Consideriamo ad esempio il problema di determinare se gli elementi di un array di interi sono tutti uguali

Algoritmo 1

- ipotizzo che gli elementi dell'array sono tutti uguali
- confronto ogni elemento con ogni altro elemento (di indice maggiore): se trovo almeno una coppia di elementi diversi, allora non è vero che sono tutti uguali

Algoritmo 2

- ipotizzo che gli elementi dell'array sono tutti uguali
- confronto ogni elemento con l'elemento successivo: se trovo almeno una coppia di elementi consecutivi diversi, allora non è vero che sono tutti uguali

Qual è la complessità dei due algoritmi ?

Complessità e Algoritmi: un esempio

Algoritmo 1

– l'operazione dominante è il confronto tra coppie di elementi

– devo confrontare ciascuno elemento dell'array con ciascun altro elemento: la complessità è **quadratica**

Algoritmo 2

– l'operazione dominante è il confronto tra coppie di elementi

– vengono confrontati solo coppie di elementi consecutivi: la complessità è **lineare**

Posso concludere che il secondo algoritmo è asintoticamente migliore del primo!!

Complessità e Algoritmi

Un problema finora irrisolto è quello dell'analisi della complessità dei metodi delle API di Java

– ad esempio, qual è la complessità dei metodi **substring** e **indexOf** della classe **String** ?

Una descrizione della complessità di tutti questi metodi non è disponibile:

– conoscere l'algoritmo usato da un certo metodo (o quelli disponibili per risolvere un certo problema) ci permette di trovare una delimitazione superiore o inferiore alla complessità del metodo;

- non è necessario conoscere il codice del singolo metodo

- la conoscenza approssimata dell'algoritmo usato è sufficiente in molti casi anche se una analisi più precisa è necessaria quando l'invocazione di un metodo può essere dominante per il metodo che dobbiamo studiare

Complessità di un programma

E' la complessità dell'algoritmo implementato dal programma.

Esempio1.

```
for(i = sum = 0; i < n; i++)
    sum += a[i];
```

Quanti enunciati di assegnamento fa l'algoritmo?

- 2 per l'inizializzazione
- 2 per ciascuna iterazione del ciclo (n iterazioni)

Totale: 2 + 2n assegnamenti.

$$T(n) = c\,(2 + 2n) \quad = O(n)$$

dimensione array di input

Esempi

Esempio2. **La complessita' di solito cresce usando cicli annidati**

```
for(i = 0; i < n; i++) {
     for (j = 1, sum = a[0]; j <= i; j++)
         sum += a[j];
     System.out.println("la somma del sotto-array da 0 a " + i
                                +" e' " + sum);
       }
```

assegnamenti

- 1 per l'inizializzazione
- 3 per ciascuna iterazione del ciclo esterno (n iterazioni)
- 2 per ciascuna iterazione del ciclo interno (i iterazioni per ogni i = 1,2, …, n-1)

$$T(n) = c\,(1 + 3n + \sum_{i=1}^{n-1} 2i) = c\,(1 + 3n + 2\,(1 + 2 + \ldots + n-1)) =$$

$$= c\,(1 + 3n + n\,(n-1)) = O(n^2)$$

dimensione array di input

Esempi

Esempio3. **La complessita' di solito cresce usando cicli annidati ma non e' sempre vero**

```
for(i = 4; i < n; i++) {
     for (j = i-3, sum = a[i-4]; j <= i; j++)
         sum += a[j];
     System.out.println("la somma del sotto-array da " + (i-4)
                                +" a " + i + " e' " + sum);
 }
```

Dai due cicli annidati, concluderemmo $T(n) = O(n^2)$.

assegnamenti

- 1 per l'inizializzazione
- 11 per ciascuna iterazione del ciclo esterno (n-4 iterazioni)

(11 = 2 + (2·4) + 1)

init. ciclo interno

2 ass. per 4 iterazioni

incremento i

$T(n) = c\,(1 + 11\,(n-4)) = O(n)$ (bound più stretto)

Esempi

Esempio4.

```
for(i = 0, length = 1; i < n-1; i++)
{ for (i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
  if(length < i2 - i1 + 1)
     length = i2 - i1 + 1;
  System.out.println("Il più lungo sotto-array ordinato è
                      lungo" + length);
}
```

Questo programma cerca le più lunghe sequenze ordinate in modo crescente in un array

Es. [1 8 1 2 5 0 11 12] la lunghezza e' 3 e corrisponde a [1 2 5]

Esempi

Esempio4.

```
for(i = 0, length = 1; i < n-1; i++)
{ for (i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
  if(length < i2 - i1 + 1)
     length = i2 - i1 + 1;
  System.out.println("Il più lungo sotto-array ordinato è
                      lungo" + length);
}
```

Il numero di assegnamenti dipende dalla forma dell'input.

- Input ordinato in senso **decrescente**: ciclo interno eseguito **1** volta per ciascuna delle n-1 iterazioni del ciclo esterno.

Allora $T_m(n) = O(n)$. Caso Migliore

Esempi

Esempio4.

```
for(i = 0, length = 1; i < n-1; i++)
{ for (i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
  if(length < i2 - i1 + 1)
     length = i2 - i1 + 1;
  System.out.println("Il più lungo sotto-array ordinato è
                      lungo" + length);
}
```

Il numero di assegnamenti dipende dalla forma dell'input.

- Input ordinato in senso **crescente**: ciclo interno eseguito **i** volte per ciascuna delle n-1 iterazioni del ciclo esterno.

Allora $T_A(n) = O(n^2)$. Caso peggiore.

E il caso medio?

Esempi

Esempio5.

```
int binarySearch(int[] arr, int key)
{ int low = 0, mid, high = arr.length-1;
  while (low <= high)
  { mid = (low + high) / 2;
    if(key < arr[mid])
      high = mid - 1;
    else if (arr[mid] < key)
           low = mid + 1;
         else return mid; // successo: key trovato in cella mid
  }
  return -1; // fallimento: key non è in arr
}
```

Anche qui il numero di assegnamenti dipende dalla forma dell'input.

Notare che l'array deve essere già ordinato

Esempi

Esempio5.

```
int binarySearch(int[] arr, int key)
{ int low = 0, mid, high = arr.length-1;
  while (low <= high)
  { mid = (low + high) / 2;
    if(key < arr[mid])
      high = mid - 1;
    else if (arr[mid] < key)
           low = mid + 1;
         else return mid; // successo: key trovato in cella mid
  }
  return -1; // fallimento: key non è in arr
}
```

- key si trova al centro dell'array: ciclo eseguito **1** volta.

Allora $T_m(n) = O(1)$. Caso Migliore

Esempi

Esempio5.

```
int binarySearch(int[] arr, int key)
{ int low = 0, mid, high = arr.length-1;
  while (low <= high)
  { mid = (low + high) / 2;
    if(key < arr[mid])
      high = mid - 1;
    else if (arr[mid] < key)
           low = mid + 1;
         else return mid; // successo: key trovato in cella mid
  }
  return -1; // fallimento: key non è in arr
}
```

- key non si trova nell'array. Sia m + 1 il numero di iterazioni del ciclo while. Ogni iterazione fa un numero costante di passi. Quindi T (n) = O(m). Vogliamo calcolare m.

Complessità di un programma

Iterazione	Dimensione sotto-array esaminato
0	n
1	$n/2$
2	$n/2^2$
3	$n/2^3$
…	…
m	$n/2^m$

All'ultima iterazione, l'array ha dimensione 1.

Quindi,

$$n/2^m = 1$$

$$m = \log_2 n$$

Complessità di un programma

$$m = \log_2 n$$

Allora $T_p(n) = O(m) = O(\log_2 n)$ Caso Peggiore

In realtà, per riconoscere i due casi come migliore o peggiore bisogna far vedere che la complessità è quella minima o massima (non sempre immediato).

Complessità – vari casi

- caso peggiore: l'algoritmo richiede tempo massimo
- caso medio: l'algoritmo richiede tempo medio
- caso migliore: l'algoritmo richiede tempo minimo

può essere complicato da calcolare

Complessità – vari casi

Esempio. Algoritmo *A* per **ordinare** in senso crescente un array di 5 interi i_1, i_2, i_3, i_4, i_5 (sia $i_1 \leq i_2 \leq i_3 \leq i_4 \leq i_5$).

Possibili input:

$input_1$ (peggiore)	i_5 i_4 i_3 i_2 i_1	*A impiega*	*passi(*$input_1$*)*
$input_2$	i_5 i_4 i_3 i_1 i_2	*A impiega*	*passi(*$input_2$*)*
$input_3$	i_5 i_3 i_4 i_1 i_2	*A impiega*	*passi(*$input_3$*)*
⋮			⋮
$input_{5!}$ (migliore)	i_1 i_2 i_3 i_4 i_5	*A impiega*	*passi(*$input_{5!}$*)*

$$\text{\# medio passi di } A = \frac{\sum_{J=1}^{5!} passi(input_J)}{5!}$$

Complessità: caso medio

Generalizzando alla generica dimensione *n* :

$$\text{\# medio passi di } A = \frac{\sum_{J=1}^{n!} passi(\text{input}_J)}{n!}$$

$$T_{me}(n) = \frac{\sum_{J=1}^{n!} passi(\text{input}_J)}{n!}$$

OK se ogni input ha probabilita' 1/n! di accadere.

Complessità: caso medio

Generalizzando :

$$T_{me}(n) = \sum_{J=1}^{n!} pr(\text{input}_J) \bullet passi(\text{input}_J)$$

Notare che il numero di elementi da considerare nella sommatoria dipende dal problema in questione.

Per esempio nel caso della ricerca binaria:

$$T_{me}(n) = \sum_{J=1}^{n} pr(\text{input}_J) \bullet passi(\text{input}_J)$$

Complessità di un problema

Finora abbiamo discusso della complessità degli *algoritmi*, cioè di procedimenti risolutivi di problemi. Tuttavia, ha senso parlare anche di complessità intrinseca dei *problemi*. Questa viene definita come segue:

Def. di complessità di un problema

Un problema ha complessità di ordine g(*n*) se qualunque algoritmo risolutivo del problema ha una complessità di ordine (g(*n*)).

In pratica, se un problema *P* ha una sua complessità g(*n*), non si potrà mai trovare un algoritmo per *P* che abbia una complessità inferiore a g(*n*).

Complessità di un Problema

Esempio: Il problema della ricerca del massimo in un array di dimensione *n* ha una complessità temporale intrinseca pari a *n*. Quindi è inutile cercare gli algoritmi che risolvono il problema in tempi proporzionali a log *n* o addirittura in tempi costanti.

Ma come determinare la complessità di un problema P?

Per dimostrare che *P* ha complessità pari a g(*n*) non basta trovare un algoritmo con complessità (g(n)). Questo, infatti, ci garantisce che il problema ha una soluzione con complessità *almeno* g(n), ma nulla impedisce che si possa trovare in futuro un altro algoritmo di complessità inferiore a g(n).

Complessità di un Problema

In generale, ogni algoritmo stabilisce un limite *superiore* per la complessità del particolare problema risolto. La determinazione di un limite *inferiore* per la complessità di un problema richiede invece la dimostrazione che *tutti* gli algoritmi risolutivi di quel problema abbiano una complessità peggiore del limite inferiore stabilito.

Esempio: Se un problema *P* fosse risolvibile mediante solo tre algoritmi, A1, A2 e A3 di complessità:

A1: $f_1(n) = n^2 \log_2 n$

A2: $f_2(n) = n \log_2 n$

A3: $f_3(n) = an$ con a>1

allora *P* sarebbe un problema di complessità $n\log_2 n$.

Complessità di un Problema

Nella pratica non possiamo pensare di valutare la complessità di un problema *P* cercando tutti gli algoritmi che risolvono *P*, perché ce ne potrebbero essere infiniti. Si deve invece ricorrere ad una vera e propria *dimostrazione* matematica, se si vogliono stabilire dei limiti *significativi* per la complessità dei problemi.

Ad esempio, si può ragionevolmente affermare che il problema di ordinare un array di *n* dati ha complessità pari almeno ad *n*, in quanto ogni algoritmo di ordinamento deve necessariamente esaminare almeno una volta ciascuno degli elementi dell'array.

Complessità di un problema

Appare comunque poco realistico pensare che si possa effettuare un ordinamento con complessità lineare rispetto ad *n*.

In effetti gli algoritmi di ordinamento hanno almeno complessità in tempo di ordine O(*n*log*n*) nel caso peggiore.

Un algoritmo che ha la stessa complessità intrinseca di un problema si dice *ottimo in ordine di grandezza*.