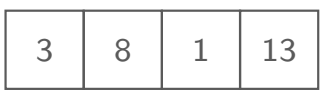
Introduzione
Computabilità e complessità

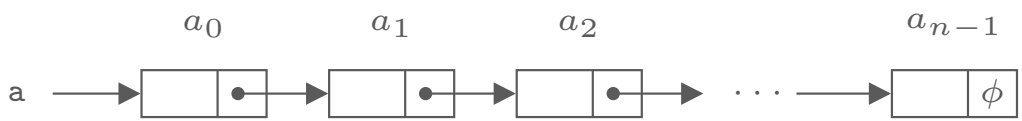
Lucidi tratti da
P. Crescenzi · G. Gambosi · R. Grossi · G. Rossi
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2012
`http://algoritmica.org`

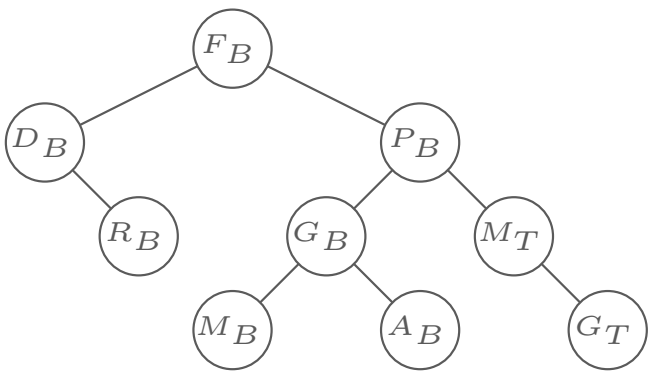
1. I lucidi sono utilizzabili dai soli docenti e se ne sconsiglia la distribuzione agli studenti: oltre al rischio di violare una qualche forma di copyright, il problema principale è che gli studenti studino in modo superficiale la materia senza il necessario approfondimento e la dovuta riflessione che la lettura del libro fornisce
2. Il simbolo [alvie] nei lucidi indica l'uso di ALVIE per visualizzare il corrispettivo algoritmo: per un proficuo rendimento dello strumento, conviene esaminare in anticipo la visualizzazione per determinare i punti salienti da mostrare a lezione (lintera visualizzazione potrebbe risultare altrimenti noiosa)

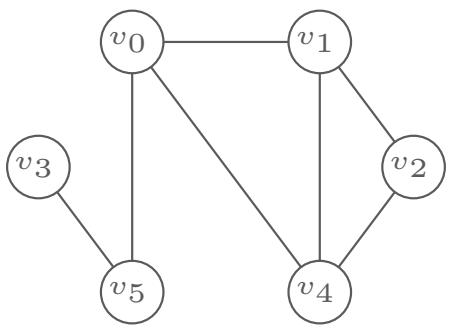
Algoritmo

- Essenza computazionale di un programma che ne descrive i passi fondamentali
- Ingredienti: **sequenze**, **alberi** e **grafi**
- Servono a strutturare i dati elementari: **bit**, **caratteri**, **interi**, **reali** e **stringhe**
- Rappresentano istanze di problemi computazionali reali e concreti









Problemi indecidibili

- Non tutti i problemi computazionali ammettono algoritmi di risoluzione: **problema della fermata** (Turing, 1937)
- Terminologia moderna: dato un generico algoritmo (o programma) A in ingresso, esso **termina** o **va in ciclo**?

Termina?

```
1 Primo( numero ):
2   fattore = 2;
3   WHILE (numero % fattore != 0)
4     fattore = fattore + 1;
5   RETURN (fattore == numero);
```

- Si, perché la variabile `fattore` è incrementata di 1 e a un certo punto deve verificare la guardia
- [alvie]

Termina?

```
1  CongetturaGoldbach( ):
2    n = 2;
3    DO {
4      n = n + 2;
5      controesempio = TRUE;
6      FOR (p = 2; p <= n-2; p = p + 1) {
7        q = n - p;
8        IF (Primo(p) && Primo(q)) controesempio = FALSE
9      }
10   } WHILE (!controesempio);
11   RETURN n;
```

- Termina se e solo se trova $n \geq 4$ per cui non esistono due primi p e q t.c. $n = p + q$
- Termina se e solo se confuta la congettura di Goldbach (problema aperto)

Problema della fermata

Non esiste un algoritmo per stabilire la terminazione di un **generico algoritmo/programma A**

1. Una sequenza di simboli può essere interpretata come **dato** o **programma**
2. Un programma può essere dato in pasto a un altro programma
3. Supponiamo che esista un algoritmo `Termina`(A, D) che, in tempo finito, restituisce SI se A termina con input D e restituisce NO se A va in ciclo con input D
4. 1+2 implicano che è legale invocare `Termina`(A, D)
5. Consideriamo il seguente algoritmo

```
1 Paradosso( A ):
2   WHILE (Termina( A, A ))
3     ;
```

6. `Paradosso( Paradosso )` termina? **CONTRADDIZIONE**

Indecidibilità

- Il problema della fermata è quindi **indecidibile**, ossia non esiste alcun algoritmo di risoluzione
- Altri problemi lo sono: stabilire l'equivalenza tra due programmi (per ogni possibile input, producono lo stesso output)

Decidibilità vs trattabilità

Problemi decidibili possono richiedere tempi di risoluzione elevati: **Torri di Hanoi**

- 3 pioli
- $n = 64$ dischi sul primo piolo (vuoti gli altri due)
- Ogni mossa sposta un disco in cima a un piolo
- Un disco non può poggiare su uno più piccolo
- Spostare tutti i dischi dal primo al terzo piolo

Soluzione ricorsiva

```
1 TorriHanoi( n, primo, secondo, terzo ):
2   IF (n = 1) {
3     PRINT primo ⟼ terzo;
4   } ELSE {
5     TorriHanoi( n - 1, primo, terzo, secondo );
6     PRINT primo ⟼ terzo;
7     TorriHanoi( n - 1, secondo, primo, terzo );
8   }
```

- [alvie]

Numero di mosse: $2^n - 1$

```
1  TorriHanoi( n, primo, secondo, terzo ):
2    IF (n = 1) {
3      PRINT primo ⟼ terzo;
4    } ELSE {
5      TorriHanoi( n - 1, primo, terzo, secondo );
6      PRINT primo ⟼ terzo;
7      TorriHanoi( n - 1, secondo, primo, terzo );
8    }
```

- Caso base $n = 1$: $2^1 - 1 = 1$
- Passo induttivo: $(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$
- 1 mossa/sec: circa 585 miliardi di anni!

Tempo esponenziale $2^n - 1$ (1 operazione/sec)

n	5	10	15	20	25	30	35	40
tempo	31 s	17 m	9 h	12 g	1 a	34 a	1089 a	34865 a

Aumentare di un fattore **moltiplicativo** X (ossia X operazioni/sec) migliora **solo** di un fattore **additivo** $\log_2 X$

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	67	70	73	77	80	83	93

Tempo polinomiale

Torri di Hanoi generalizzate con $k > 3$ pioli

- Pioli numerati da 0 a $k - 1$
- Ipotesi semplificativa: n è multiplo di $k - 2$

```
1  TorriHanoiGen( n, k ):
2    FOR (i = 1; i <= k-2; i = i+1)
3      TorriHanoi(n/(k-2), 0, k-1, i);
4    FOR (i = k-2; i >= 1; i = i-1)
5      TorriHanoi(n/(k-2), i, 0, k-1);
```

Torri di Hanoi generalizzate

- Il codice richiede $2(k-2)(2^{n/(k-2)}-1)$ mosse
- Al più n^2 mosse, fissando $k = \lfloor n/\log n \rfloor$ e $n \geq 5$
- $n = 64$: al più $64^2 = 4096$ mosse

```
1 TorriHanoiGen( n, k ):
2   FOR (i = 1; i <= k-2; i = i+1)
3     TorriHanoi(n/(k-2), 0, k-1, i);
4   FOR (i = k-2; i >= 1; i = i-1)
5     TorriHanoi(n/(k-2), i, 0, k-1);
```

Tempo esponenziale n^2 (1 operazione/sec)

n	5	10	15	20	25	30	35	40
tempo	25 s	100 s	225 s	7 m	11 m	15 m	21 m	27 m

Aumentare di un fattore **moltiplicativo** X (ossia X operazioni/sec) migliora di un fattore **moltiplicativo** $\sqrt{X}$

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6
numero dischi	64	202	640	2023	6400	20238	64000

Dimensione n dei dati per un problema generico

- **Numero di bit**: k bit possono rappresentare interi in $\{0, 1, \ldots, 2^k - 1\}$

  $b_{k-1} \cdots b_1 b_0$ rappresenta $n = \sum_{i=0}^{k-1} b_i \times 2^i$

  Ad esempio, per $k = 3$

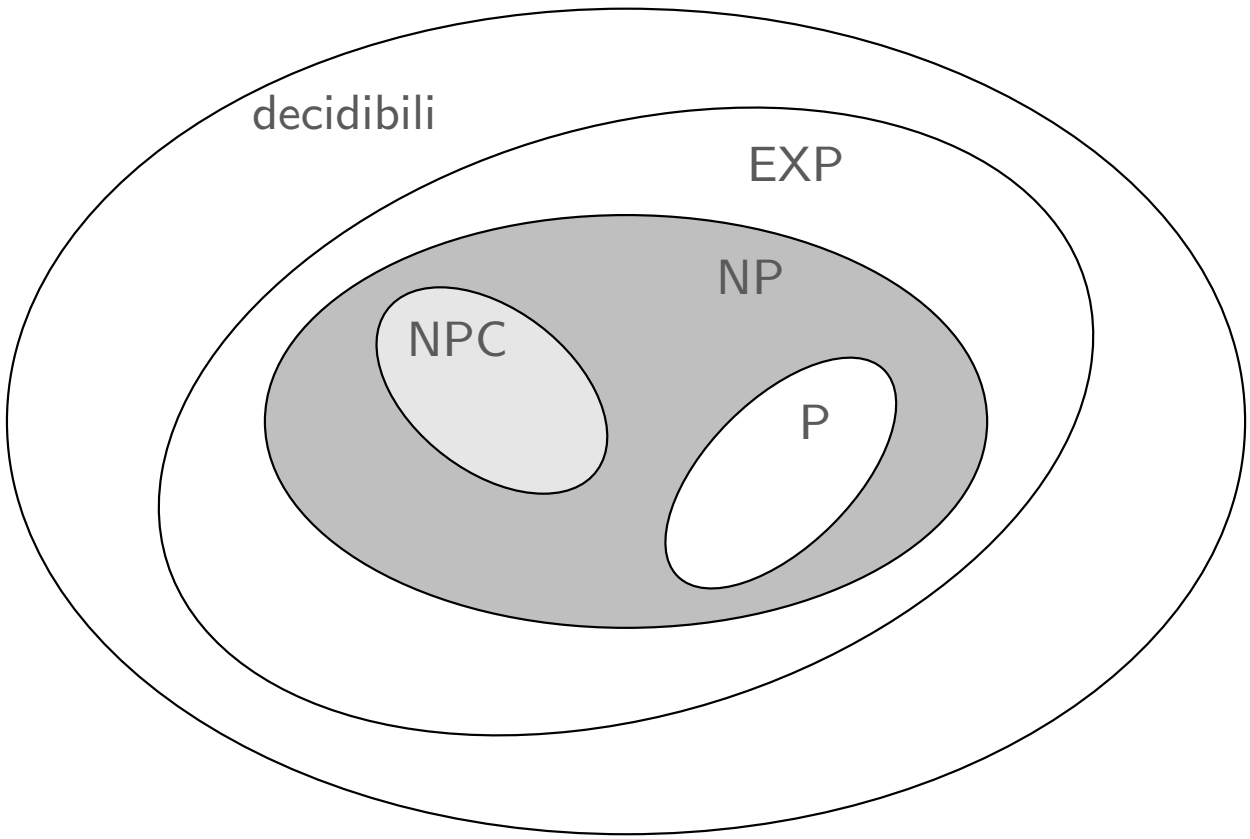
000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

  Caratteri: 8 bit (ASCII) o 16 bit (Unicode/UTF8)
  Reali: 32, 64 o 128 bit (segno, esponente, mantissa)
- **Numero di elementi**: array, stringhe, liste, insiemi
- **Numero di celle di memoria** occupate dai dati (ciascuna contenente $O(\log n)$ bit)

Algoritmo polinomiale

Esistono due costanti $c, n_0 > 0$ *tali che il numero di passi elementari è al più* n^c *per ogni input di dimensione* n *e per ogni* $n > n_0$

- Problemi **trattabili**: esiste algoritmo polinomiale
- Problemi intrattabili: non esiste algoritmo polinomiale



Classi di complessità

- **P** = classe dei problemi risolvibili deterministicamente in tempo polinomiale
- **EXP** = classe dei problemi risolvibili deterministicamente in tempo esponenziale
- P: problemi trattabili
- EXP: utile solo per piccole istanze di problemi

Generazione delle 2^n sequenze binarie

- Equivale a generare ricorsivamente tutti i sotto-insiemi di un insieme di n elementi ($A[i] = 1$ se e solo se l'i-esimo elemento è selezionato)
  Ad esempio, per $n = 3$

000	001	010	011	100	101	110	111
$\emptyset$	{3}	{2}	{2, 3}	{1}	{1, 3}	{1, 2}	{1, 2, 3}

- Struttura ricorsiva della generazione
  000, 100, 010, 110, 001, 101, 011, 111
  000, 100, 010, 110
  000

fissa il bit a 1 e ricorri come per il bit a 0

Generazione delle 2^n sequenze binarie

- Equivale a generare ricorsivamente tutti i sotto-insiemi di un insieme di n elementi ($A[i] = 1$ se e solo se l'i-esimo elemento è selezionato)

```
1  GeneraBinarie( A, b ):
2    IF (b == 0) {
3      Elabora( A );
4    } ELSE {
5      A[b-1] = 0;
6      GeneraBinarie( A, b-1 );
7      A[b-1] = 1;
8      GeneraBinarie( A, b-1 );
9    }
```

- Invocato con $b = n$
- [alvie]

Generazione delle $n!$ permutazioni di A

a b c d	a b d c	a d c b	d b c a
b a c d	b a d c	d a c b	b d c a
a c b d	a d b c	a c d b	d c b a
c a b d	d a b c	c a d b	c d b a
c b a d	d b a c	c d a b	c b d a
b c a d	b d a c	d c a b	b c d a
$i = 3$	$i = 2$	$i = 1$	$i = 0$

Per $i = n - 1, \ldots, 1, 0$:

- scambia $A[i]$ con quello in ultima posizione, ovvero con $A[n-1]$
- i primi $n - 1$ elementi di A sono ricorsivamente permutati **nella stessa maniera** (indipendentemente da i)
- scambia lultimo elemento $A[n-1]$ con $A[i]$ (per rimetterli a posto)

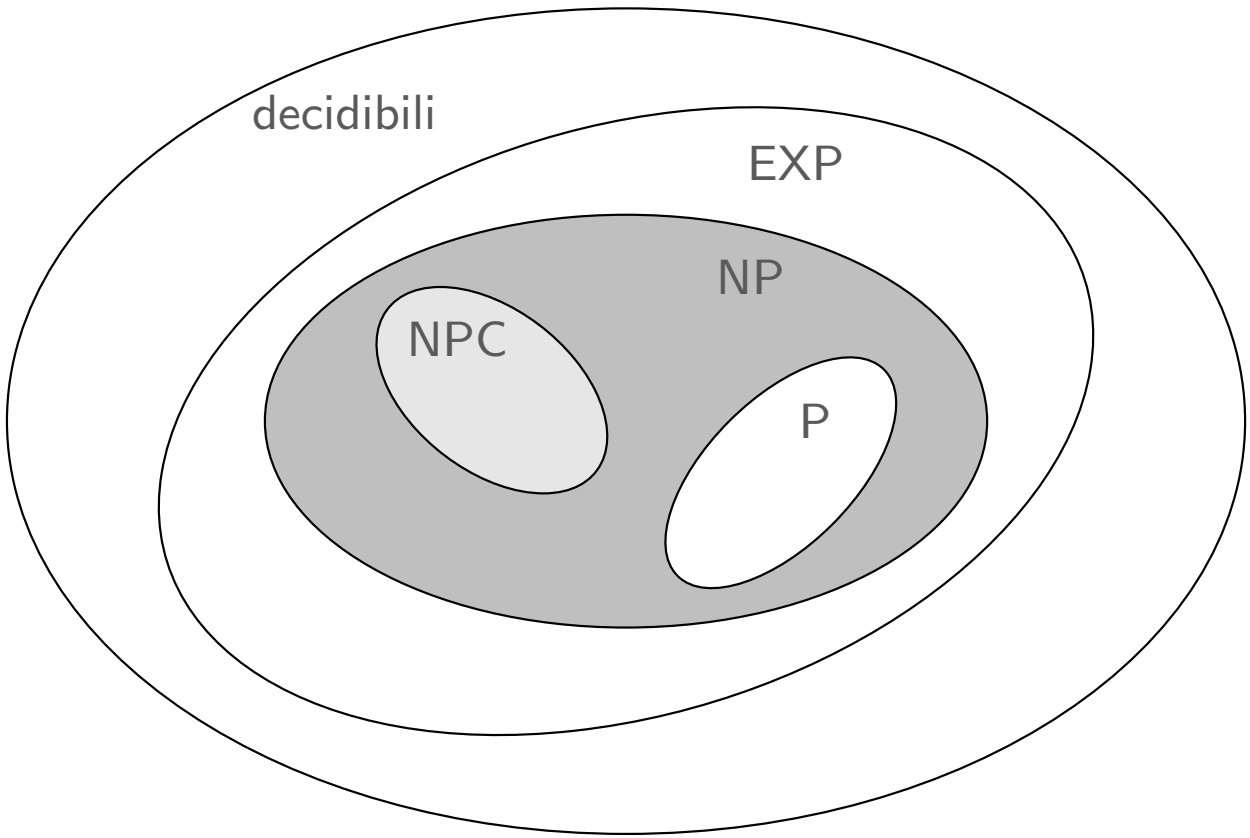
Generazione delle $n!$ permutazioni di A

```
1  GeneraPermutazioni( A, p ):
2    IF (p == 0) {
3      Elabora( A );
4    } ELSE {
5      FOR (i = p-1; i >= 0; i = i-1) {
6        Scambia( i, p-1 );
7        GeneraPermutazioni( A, p-1 );
8        Scambia( i, p-1 );
9      }
10   }
```

- Invocato con $p = n$
- [alvie]

Zona "grigia": classi NP e NPC

- **NP** = classe dei problemi risolvibili **nondeterministicamente** in tempo polinomiale
- **NPC** = classe dei problemi completi per NP, detti **NP-completi**



Sudoku

- Tabella 9×9 contenente numeri compresi tra 1 e 9
- Divisa in 3×3 sottotabelle (di taglia 3×3)
- Alcune celle contengono numeri, altre vuote
- Riempire le celle vuote in modo che
  1. Ogni riga contiene una permutazione di $1, 2, \ldots, 9$
  2. Ogni colonna contiene una permutazione di $1, 2, \ldots, 9$
  3. Ogni sottotabella contiene una permutazione di $1, 2, \ldots, 9$

Sudoku

3	9							8
	7	1			3			
		8		4	9		6	
1			2	7				9
6								3
5				3	6			4
	4		1	5		9		
			9			8	2	
9							4	7

3	9	6	5	1	2	4	7	8
4	7	1	6	8	3	5	9	2
2	5	8	7	4	9	3	6	1
1	3	4	2	7	5	6	8	9
6	8	7	4	9	1	2	5	3
5	2	9	8	3	6	7	1	4
8	4	2	1	5	7	9	3	6
7	1	3	9	6	4	8	2	5
9	6	5	3	2	8	1	4	7

- Soluzione ottenibile in questo caso attraverso implicazioni logiche
- Ad esempio, nella sottotabella in alto a destra il 3 può stare solo in basso a sinistra

Backtrack con scelte non uniche

			6		2		9	
								6
			7	3	1	5		8
4		9	3			6		5
		3				1		
5		8			7	9		2
		1	5	2	3			
7								
	6	2	9		4			

			6		2		9	
			8					6
			7	3	1	5		8
4	2	9	3	1	8	6	7	5
6	7	3	2			1	8	4
5	1	8	4	6	7	9	3	2
		1	5	2	3			
7			1	8	6			
	6	2	9	7	4			

Partendo dalla configurazione a sinistra, giungiamo nella configurazione a destra che ammette diverse scelte per ogni casella

Backtrack: *algoritmo che esplora tali scelte, annullando gli effetti nel caso che non conducano a soluzione*

[alvie]

Backtrack per Sudoku

- Esamina le m caselle vuote nell'ordine indicato da `PriVuo`, `SucVuo`, `UltVuo`

```
1  Sudoku( c ):
2    elenco = insieme cifre ammissibili per c;
3    FOR (i = 0; i < |elenco|; i = i+1) {
4      Assegna( c, elenco[i] );
5      IF (!UltVuo(c) && !Sudoku(SucVuo(c))) {
6        Svuota( c );
7      } ELSE {
8        RETURN TRUE;
9      }
10   }
11   RETURN FALSE;
```

  Invocata con c = `PriVuo()`

- Nel caso pessimo, esplora circa 9^m scelte $(m \leq 9^2)$
- In generale, tabella $n \times n$: circa $n^m \leq n^{n^2} = 2^{n^2 \log n}$

Sudoku: quale complessità?

- L'algoritmo di backtrack è quindi esponenziale, ma il Sudoku $n \times n$ è trattabile o meno?
- Dipende dall'esistenza di un algoritmo polinomiale: a oggi, tale algoritmo è ignoto
- Sudoku sembra avere una natura computazionale diversa da quella delle Torri di Hanoi: possiamo verificare la correttezza di una soluzione in tempo polinomiale (cosa non possibile con le Torri di Hanoi)

Sudoku: verifica polinomiale di una soluzione

```
1  VerificaSudoku( sequenza ):
2    casella = PriVuo( );
3    FOR (i = 0; i < m; i = i+1) {
4      c = sequenza[i];
5      IF (c in casella.riga) RETURN FALSE;
6      IF (c in casella.colonna) RETURN FALSE;
7      IF (c in casella.sotto_tabella) RETURN FALSE;
8      Assegna( casella, c );
9      casella = SucVuo(casella);
10   }
11   RETURN TRUE;
```

- Richiede circa $m \times n \leq n^3$ passi (ordine di crescita)
- Sudoku è uno delle migliaia di problemi NPC: *possiamo verificare ogni sua soluzione con un algoritmo polinomiale; sappiamo soltanto trovarla con un algoritmo esponenziale (e non si sa se ne esiste uno polinomiale)*

Problemi in NP (definizione informale)

- **Certificato polinomiale** per un problema computazionale Π:
  - chi ha la soluzione per un'istanza di Π, può convincerci di ciò in tempo polinomiale
  - chi non ha tale soluzione, deve procedere per tentativi mediante backtrack esponenziale
- **NP** = classe dei problemi che ammettono un certificato polinomiale

Osservazione. $P \subseteq NP$ (basta certificato nullo se $\Pi \in$ P)

Problemi NP-completi: **NPC** (definizione informale)

- NPC $\subseteq$ NP e non si sa se tali problemi siano trattabili
- Ogni problema $\Pi \in$ NP può essere ricondotto a un problema $\Sigma \in$ NPC attraverso una **riduzione polinomiale** ($\Pi \leq \Sigma$)
- Di conseguenza:
  - se un problema in NPC è **trattabile**, allora tutti lo sono in NPC e vale $\mathbf{P = NP}$
  - se un problema in NPC è **intrattabile**, allora tutti lo sono in NPC e vale $\mathbf{P \neq NP}$
- **P=NP?** è un famoso problema aperto in informatica (definizioni più rigorose a fine corso)

Modello di calcolo RAM (Random Access Machine)

- Schema di von Neumann: dati e programmi sono sequenze binarie contenute nella memoria
- Caratteristiche principali:
  - contatore di programma e registro accumulatore
  - memoria di dimensione illimitata
  - processore esegue operazioni aritmetiche, di confronto, logiche, di trasferimento e di controllo
- **Costo uniforme delle operazioni**: costante e non dipende dalla dimensione n dei dati

Analisi di complessità

- **Costo di un algoritmo** è in funzione di n (dimensione dei dati in input):
  - **tempo** = numero di operazioni RAM eseguite
  - **spazio** = numero di celle di memoria occupate (escluse quelle per contenere l'input)
- Notazione asintotica al crescere di n:
  - $\mathbf{g(n) = O(f(n))}$ se solo se $\exists c, n_0 > 0 : \mathbf{g(n)} \leq \mathbf{cf(n)}\ \forall n > n_0$
  - $\mathbf{g(n) = \Omega(f(n))}$ se solo se $\exists c, n_0 > 0 : \mathbf{g(n)} \geq \mathbf{cf(n)}$ per infiniti valori $n > n_0$
  - $\mathbf{g(n) = \Theta(f(n))}$ se solo se $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$

Caso pessimo e medio

Complessità o costo computazionale $f(n)$ in tempo e in spazio di un problema Π:

- **caso pessimo** o **peggiore** $=$ costo **max** tra tutte le istanze di Π aventi dimensioni dei dati pari a n
- **caso medio** $=$ costo **mediato** tra tutte le istanze di Π aventi dimensioni pari a n

Guida per il calcolo del costo al caso pessimo

- `IF (guardia) { blocco1 } ELSE { blocco2 }`

$$\text{costo}(\texttt{guardia}) + \max\{\text{costo}(\texttt{blocco1} + \text{costo}(\texttt{blocco2})\}$$

- `FOR (i = 0; i < m; i = i + 1) { corpo }`

$$\sum_{i=0}^{m-1} t_i$$

dove t_i è il costo di `corpo` all'iterazione i

- `WHILE (guardia) { corpo }`

$$\sum_{i=0}^{m} (t'_i + t_i)$$

dove m sono le volte in cui `guardia` è soddisfatta, t'_i è il costo di `guardia` all'iterazione i, t_i è il costo di `corpo` all'iterazione i

Guida per il calcolo del costo al caso pessimo

- Il costo di una chiamata a funzione è il costo del suo corpo più il passaggio dei parametri (le funzioni ricorsive saranno trattate in seguito)
- Il costo di una sequenza di istruzioni è la somma dei costi delle istruzioni nella sequenza
- Applicheremo implicitamente queste semplici regole nel resto del libro