

**Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base**  
**Corso di Laurea Magistrale in Ingegneria Informatica**



# **Corso di Algoritmi e Strutture Dati**

Analisi di correttezza e di complessità degli algoritmi



# Correttezza



DIE  
TI.  
UNI  
NA

- **Problema**
- Data una sequenza di  $n$  valori di ingresso  $(a_1, a_2, \dots, a_n)$ , determinare una permutazione  $(a'_1, a'_2, \dots, a'_n)$  della sequenza di ingresso tale che  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ )

## • **Applicazioni**

- Searching
- Closest pair
- Element uniqueness
- Frequency distribution
- Selection in an array
- Convex hulls
- ...

*Sorting lies at the heart of many algorithms. Sorting the data is one of the first things any algorithm designer should try in the quest for efficiency.*

# Insertion Sort



- Insertion Sort è un algoritmo che considera un valore alla volta e lo inserisce nella corretta posizione tra i valori che lo precedono
- È un algoritmo che ordina i valori **sul posto**

Spazio in memoria al massimo di quanto è grande il vettore.
- I valori sono riordinati all'interno dell'array stesso, con al più un numero costante di essi memorizzati al di fuori dell'array in ogni momento

# Insertion Sort



<https://www.i-programmer.info/programming/theory/3531-sorting-algorithms-as-dances.html>

<https://visualgo.net/en/sorting>

# Insertion Sort



2 5 2 4 6 1 3  
4 2 5 4 6 1 3  
6 2 4 5 6 1 3  
1 2 4 5 6 1 3  
3 1 2 4 5 6 3

```
Insertion-Sort (A)
for j < 2 to length[A]
    do key <- A[j]
        // Insert A[j] into A[1..j-1]
        i <- j-1
        while i>0 and A[i]>key
            do A[i+1] <- A[i]
            i <- i-1
        A[i+1] <- key
```

# Insertion Sort



2	5	2	4	6	1	3
key	i	j				
4	2	5	4	6	1	3
6	2	4	5	6	1	3
1	2	4	5	6	1	3
3	1	2	4	5	6	3

```
Insertion-Sort (A)
for j < 2 to length[A]
    do key <- A[j]
        // Insert A[j] into A[1..j-1]
        i <- j-1
        while i>0 and A[i]>key
            do A[i+1] <- A[i]
            i <- i-1
        A[i+1] <- key
```

# Insertion Sort



- Per dimostrare la correttezza, utilizziamo **l'invariante**:
- *All'inizio di ciascuna iterazione del ciclo for, la sottosequenza A[1..j-1] consiste degli elementi originali di A[1..j-1] ma ordinati in senso crescente*
- Occorre dimostrare che:

Caso base

1. L'invariante è vero prima di iniziare il ciclo

Caso n

2. Se l'invariante è vero prima di una iterazione del ciclo, rimane vero prima della successiva iterazione

Caso n +1

3. Quando il ciclo termina, l'invariante prova che l'algoritmo è corretto

# Insertion Sort



- Invariante:
- *All'inizio di ciascuna iterazione del ciclo for, la sottosequenza A[1..j-1] consiste degli elementi originali di A[1..j-1] ma ordinati in senso crescente*
- Ne deriva che:
- L'invariante è vero prima della prima iterazione
- Per j=2, la sottosequenza A[1..j-1] è costituita solo da A[1], che è ovviamente una sequenza ordinata
- Una iterazione del ciclo conserva la verità dell'invariante
- L'elemento A[j] viene inserito in modo da mantenere la sequenza A[1..j] ordinata
- L'invariante prova che l'algoritmo è corretto
- Il ciclo termina quando j=n+1, ovvero quando la sequenza A[1..n] è ordinata

# Convenzioni per lo pseudocodice

- L'indentazione indica la struttura dei blocchi (no parentesi)
- I costrutti iterativi e di condizione hanno significato analogo ai linguaggi di alto livello
- La notazione  $i \leftarrow j$  indica che il valore di  $j$  viene assegnato a  $i$
- L'operatore  $[]$  indica sia l'accesso ai singoli elementi di un array che l'accesso ai campi di un oggetto (es.  $\text{length}[A]$ )
- I parametri sono passati per valore alle procedure
- Gli operatori booleani `and` e `or` sono valutati secondo la tecnica del corto circuito (come nella maggior parte dei linguaggi di programmazione)

- Mostrare l'«incorrettezza»
  - «**Searching for **counterexamples** is the best way to disprove the correctness**»
- *Testing*
  - Definire casi di test aiuta scoprire malfunzionamenti, dimostrando l'incorrettezza dell'algoritmo
  - Da usare in congiunzione con l'analisi formale di correttezza

## . Tips

- **Think small:** *there is usually a very simple example on which alg. fail*
- **Think exhaustively:** *there are only a small number of possibilities for the smallest nontrivial value of n; all cases, inclugin counter-examples, can be constructed by these possibilities*
- **Hunt for the weakness:** e.g., greedy algorithms (i.e.: of the form “always take the biggest”) can be broken by the next «go for a tie» tip
- **Go for a tie:** e.g., for greedy alg, provide instances where everything is the same size
- **Seek extremes** (*e.g. boundary-value testing*): *mixtures of huge and tiny, left and right, few and many, near and far*



## S&T: Factorial Formulae

**Problem:** Prove that  $\sum_{i=1}^n i \times i! = (n + 1)! - 1$  by induction.

Tip: Dimostrare la correttezza di sommatorie è una applicazione classica di induzione (torna utile nell'analisi degli algoritmi)  
*Nel passo induttivo, si suggerisce di scomporre la sommatoria*



## S&T: Incremental Correctness

**Problem:** *Prove the correctness of the following recursive algorithm for incrementing natural numbers, i.e.  $y \rightarrow y + 1$ :*

```
Increment(y)
  if y = 0 then return(1) else
    if (y mod 2) = 1 then
      return(2 · Increment([y/2]))
    else return(y + 1)
```

**Tip:** Induzione e ricorsione sono concetti chiave per l'analisi (e la progettazione) degli algoritmi

*Passo induttivo: si suggerisce di usare  $y \leq n-1$  (e dimostrare che funziona per  $y=n$ ) piuttosto che  $y=n-1$*

# Analisi di complessità



# Analisi di un algoritmo

- Analizzare un algoritmo significa stimare le risorse richieste in termini di:
  - **Occupazione di memoria**
  - **Impegno di banda**
  - **Tempo di calcolo**
- Occorre definire il modello del calcolatore usato per eseguire l'algoritmo
  - **Singolo processore**
  - **Memoria ad accesso casuale**
  - **Assenza di gerarchie di memoria**
- Assunzioni:
  - Le istruzioni aritmetiche, di accesso alla memoria e di controllo richiedono un tempo costante

# Analisi di un algoritmo

- Il tempo di esecuzione di un algoritmo è solitamente espresso in funzione della dimensione dei valori in ingresso
  - Per algoritmi di ordinamento: lunghezza della sequenza da ordinare
  - Per algoritmi che operano su grafi: numero di vertici e numero di archi
- Il tempo di esecuzione è dato dalla somma dei tempi di esecuzione di ciascuna linea dello pseudocodice (che assumiamo essere costanti), ciascuno moltiplicato per il numero di volte che la linea viene eseguita

# Analisi di Insertion Sort



DIE  
TI.  
UNI  
NA

Insertion-Sort (A)

```
for j < 2 to length[A]
    do key <- A[j]
        // Insert A[j] into A[1..j-1]
        i <- j-1
        while i>0 and A[i]>key
            do A[i+1] <- A[i]
            i <- i-1
        A[i+1] <- key
```

*cost*

$c_1$

$c_2$

$c_4$

$c_5$

$c_6$

$c_7$

$c_8$

*times*

$n$

$n-1$

$n-1$

$\sum_{j=2}^n t_j$

$\sum_{j=2}^n (t_j - 1)$

$\sum_{j=2}^n (t_j - 1)$

$n-1$

$t_j$  è il numero di volte che viene eseguito il test del ciclo while

# Analisi di Insertion Sort



$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1)$$

- Il caso migliore si verifica quando il vettore è già ordinato ( $t_j=1$ )

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) = an + b$$

- Il tempo di esecuzione è **funzione lineare** di n
- Il caso peggiore si verifica quando il vettore è ordinato in senso decrescente ( $\sum_{2 \leq j \leq n} j = n(n+1)/2 - 1$ ,  $\sum_{2 \leq j \leq n} (j-1) = n(n-1)/2$ )

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \left( \frac{n(n + 1)}{2} - 1 \right) + c_6 \left( \frac{n(n - 1)}{2} \right) + c_7 \left( \frac{n(n - 1)}{2} \right) + c_8(n - 1)$$
$$T(n) = an^2 + bn + c$$

- Il tempo di esecuzione è **funzione quadratica** di n

# Analisi di Insertion Sort

- Tipicamente si considera il tempo di esecuzione nel **caso peggiore**
  - È un limite superiore per il tempo di esecuzione di qualsiasi istanza
  - Per alcuni algoritmi, il caso peggiore si presenta piuttosto spesso
  - Il caso “medio” non è tipicamente molto migliore del caso peggiore
- Per indicare sinteticamente il tempo di esecuzione di un algoritmo, si tralasciano le costanti e si indica solo il termine dominante
  - Insertion Sort ha un tempo di esecuzione nel caso peggiore di  $\Theta(n^2)$



# Insertion Sort: implementazione in C++

Insertion-Sort (A)

```
for j < 2 to length[A]
    do key < A[j]
        // Insert A[j] into A[1..j-1]
        i < j-1
        while i>0 and A[i]>key
            do A[i+1] <- A[i]
            i <- i-1
        A[i+1] <- key
```

```
template <typename T>
void InsertionSort (vector<T> A)
{
    for (int j=1; j<A.size(); j++)
    {
        T key = A[j];
        // Insert A[j] into A[1..j-1]
        int i = j-1;
        while (i>=0 && A[i]>key)
        {
            A[i+1] = A[i];
            i--;
        }
        A[i+1] = key;
    }
}
```

# Notazione asintotica e crescita delle funzioni



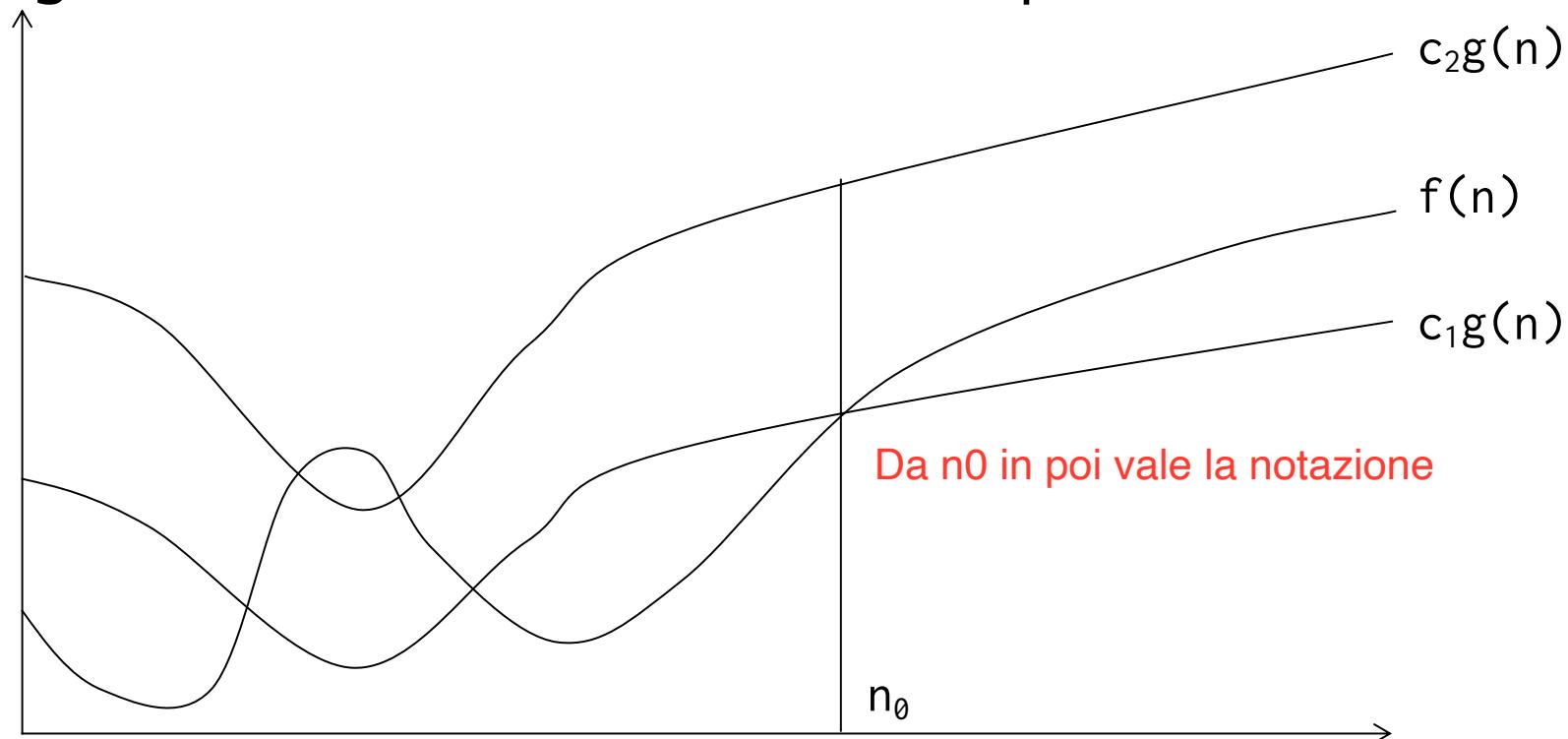
# Tempo di esecuzione asintotico



- Caratterizziamo l'efficienza di un algoritmo mediante il tempo di esecuzione dell'algoritmo
- Quando la dimensione dei dati in ingresso è tale da rendere rilevante solo l'*ordine di crescita* del tempo di esecuzione, stiamo studiando l'efficienza **asintotica** di un algoritmo

# Notazione $\Theta$

- Data la funzione  $g(n)$ ,  $\Theta(g(n))$  denota l'*insieme* di funzioni:
- $\Theta(g(n)) = \{f(n): \text{esistono costanti positive } c_1, c_2 \text{ e } n_0 \text{ tali che } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ per ogni } n \geq n_0\}$
- Usiamo (impropriamente)  $f(n) = \Theta(g(n))$  anziché  $f(n) \in \Theta(g(n))$
- $g(n)$  è un limite asintotico stretto per  $f(n)$



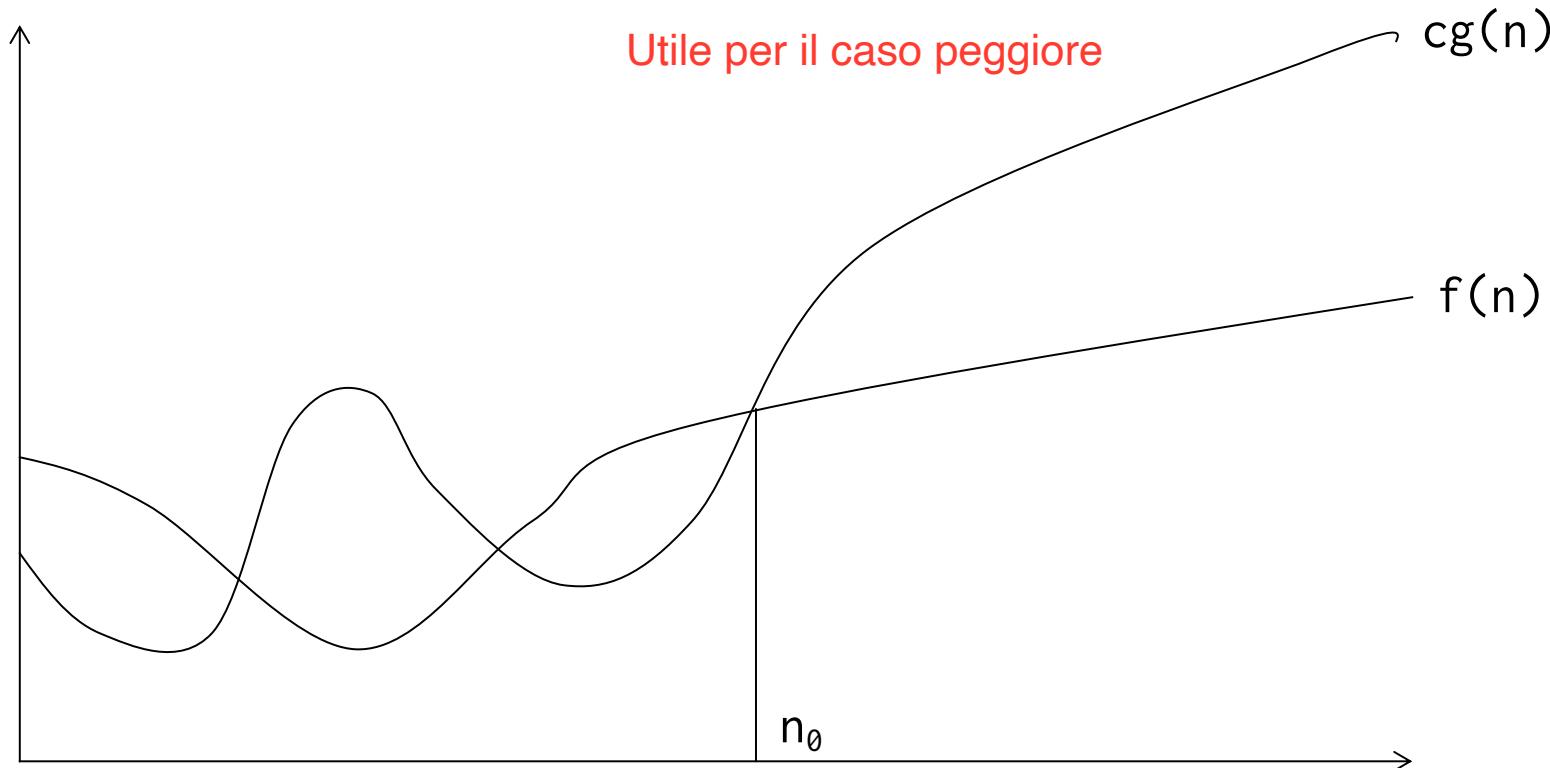
# Notazione $\Theta$



- Nel calcolo del limite asintotico è possibile ignorare i termini della funzione di ordine inferiore
- es.  $\frac{1}{2} n^2 - 3n = \Theta(n^2)$
- Per verificarlo, occorre trovare  $c_1, c_2$  e  $n_0$  tali che  $c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$  per ogni  $n \geq n_0$
- Dividendo per  $n^2$ :  $c_1 \leq \frac{1}{2} - 3/n \leq c_2$  per ogni  $n \geq n_0$
- La diseguaglianza a destra vale per  $n \geq 1$  scegliendo  $c_2 \geq \frac{1}{2}$
- La diseguaglianza a sinistra vale per  $n \geq 7$  scegliendo  $c_1 \leq 1/14$
- La definizione è verificata con  $c_1 = 1/14$ ,  $c_2 = \frac{1}{2}$  e  $n_0 = 7$

# Notazione O

- Data la funzione  $g(n)$ ,  $O(g(n))$  denota l'*insieme* di funzioni:
- $O(g(n)) = \{f(n): \text{esistono costanti positive } c \text{ e } n_0 \text{ tali che } 0 \leq f(n) \leq cg(n) \text{ per ogni } n \geq n_0\}$
- $g(n)$  è un limite superiore asintotico per  $f(n)$



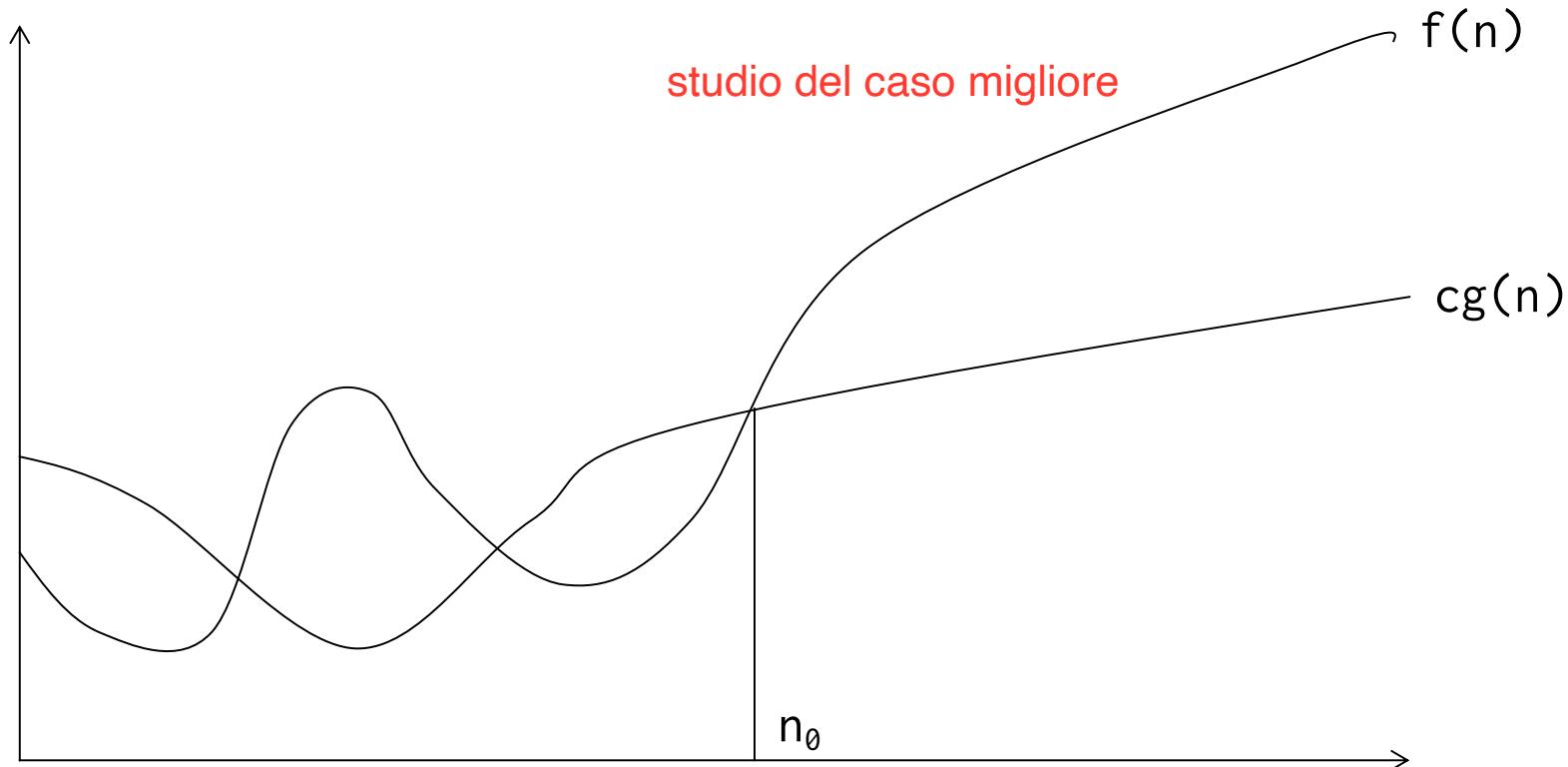
# Notazioni O e Θ



- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
- $\Theta(g(n)) \subseteq O(g(n))$
- $an+b \neq \Theta(n^2)$  ma  $an+b = O(n^2)$
- Se il tempo di esecuzione di un algoritmo nel caso peggiore è  $O(g(n))$ , il tempo di esecuzione per *qualunque* ingresso è sempre  $O(g(n))$
- La notazione O fornisce un limite superiore
- Se il tempo di esecuzione di un algoritmo nel caso peggiore è  $\Theta(g(n))$ , non è detto che il tempo di esecuzione per *qualunque* ingresso sia  $\Theta(g(n))$
- es.  
InsertionSort: caso peggiore  $\Theta(n^2)$ , sequenza ordinata  $\Theta(n)$

# Notazione $\Omega$

- Data la funzione  $g(n)$ ,  $\Omega(g(n))$  denota l'*insieme* di funzioni:
- $\Omega(g(n)) = \{f(n): \text{esistono costanti positive } c \text{ e } n_0 \text{ tali che } 0 \leq cg(n) \leq f(n) \text{ per ogni } n \geq n_0\}$
- $g(n)$  è un limite inferiore asintotico per  $f(n)$



# Notazioni asintotiche



- $f(n)=\Theta(g(n)) \Leftrightarrow f(n)=O(g(n))$  e  $f(n)=\Omega(g(n))$
- Se il tempo di esecuzione di un algoritmo nel caso migliore è  $\Omega(g(n))$ , il tempo di esecuzione per *qualunque* ingresso è sempre  $\Omega(g(n))$
- La notazione  $\Omega$  fornisce un limite inferiore
- Il tempo di esecuzione di Insertion Sort è  $\Omega(n)$  e  $O(n^2)$

# Notazioni asintotiche



- Quando una notazione asintotica appare in una equazione va interpretata come una funzione ignota che non ci interessa specificare
  - es.  $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$  equivale a  $2n^2 + 3n + 1 = 2n^2 + f(n)$ , dove  $f(n)$  è una qualunque funzione nell'insieme  $\Theta(n)$
- Proprietà
- **Transitiva** (valida per  $O$ ,  $\Theta$ ,  $\Omega$ )
  - $f(n) = O(g(n))$  e  $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- **Riflessiva** (valida per  $O$ ,  $\Theta$ ,  $\Omega$ )
  - $f(n) = O(f(n))$
- **Simmetrica**
  - $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
- **Anti-simmetrica**
  - $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

# Crescita delle funzioni



$n$	$f(n)$	$\lg n$	$n$	$n \lg n$	$n^2$	$2^n$	$n!$
10		0.003 $\mu$ s	0.01 $\mu$ s	0.033 $\mu$ s	0.1 $\mu$ s	1 $\mu$ s	3.63 ms
20		0.004 $\mu$ s	0.02 $\mu$ s	0.086 $\mu$ s	0.4 $\mu$ s	1 ms	77.1 years
30		0.005 $\mu$ s	0.03 $\mu$ s	0.147 $\mu$ s	0.9 $\mu$ s	1 sec	$8.4 \times 10^{15}$ yrs
40		0.005 $\mu$ s	0.04 $\mu$ s	0.213 $\mu$ s	1.6 $\mu$ s	18.3 min	
50		0.006 $\mu$ s	0.05 $\mu$ s	0.282 $\mu$ s	2.5 $\mu$ s	13 days	
100		0.007 $\mu$ s	0.1 $\mu$ s	0.644 $\mu$ s	10 $\mu$ s	$4 \times 10^{13}$ yrs	
1,000		0.010 $\mu$ s	1.00 $\mu$ s	9.966 $\mu$ s	1 ms		
10,000		0.013 $\mu$ s	10 $\mu$ s	130 $\mu$ s	100 ms		
100,000		0.017 $\mu$ s	0.10 ms	1.67 ms	10 sec		
1,000,000		0.020 $\mu$ s	1 ms	19.93 ms	16.7 min		
10,000,000		0.023 $\mu$ s	0.01 sec	0.23 sec	1.16 days		
100,000,000		0.027 $\mu$ s	0.10 sec	2.66 sec	115.7 days		
1,000,000,000		0.030 $\mu$ s	1 sec	29.90 sec	31.7 years		

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \lg n \gg n \gg \log n \gg 1$$

**Take-Home Lesson.** «Although esoteric functions arise in advanced algorithm analysis, a small variety of time complexities suffice and account for most algorithms that are widely used in practice.»



DIE  
TI. UNI  
NA



S&T:  
*Implementa  
ed analizza*

Esercizio  
Libro di testo: 2.2.2



## Pseudo-code

- *Find the smallest element. Swap it with the first element.*
- *Find the second-smallest element. Swap it with the second element.*
- *Repeat finding the next-smallest element, and swapping it into the correct position for the first  $n-1$  elements*
- **Implementa**
- **Analizza:**
  - *Quale invariante di ciclo conserva?*
  - *perché bisogna eseguirlo fino ad  $n-1$  anziché  $n$ ?*
  - *Esprimere nella notazione  $\Theta$  i tempi di esecuzione migliore e peggiore*

# Crescita delle funzioni: ragionare sull'efficienza



```
selection_sort(int s[], int n) {  
    int i,j;          /* counters */  
    int min;          /* index of minimum */  
    for (i=0; i<n; i++) {  
        min=i;  
        for (j=i+1; j<n; j++)  
            if (s[j] < s[min]) min=j;  
        swap(s[i],s[min]);  
    }  
}
```

Valutare l'efficienza asintotica

# Crescita delle funzioni: ragionare sull'efficienza



## S&T: String Pattern Matching

**Problem:** Substring Pattern Matching

**Input:** A text string  $t$  and a pattern string  $p$ .

**Output:** Does  $t$  contain the pattern  $p$  as a substring, and if so where?

a	b					
a	b	b				
a						
—————						
a	a	b	a	b	b	a

Searching for the substring  $abba$  in the text  $aababba$ .

Valutare l'efficienza asintotica

# Crescita delle funzioni: ragionare sull'efficienza



## S&T: String Pattern Matching

**Problem:** Substring Pattern Matching

**Input:** A text string  $t$  and a pattern string  $p$ .

**Output:** Does  $t$  contain the pattern  $p$  as a substring, and if so where?

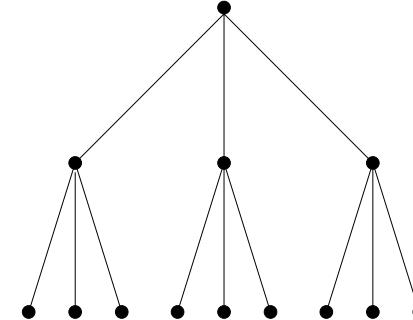
Valutare  
l'efficienza  
asintotica

```
int findmatch(char *p, char *t) {  
    int i, j;          /* counters */  
    int m, n;          /* string lengths */  
    m = strlen(p);  
    n = strlen(t);  
    for (i=0; i<=(n-m); i=i+1) {  
        j=0;  
        while ((j<m) && (t[i+j]==p[j]))  
            j = j+1;  
        if (j == m) return(i);  
    }  
    return(-1);  
}
```

# Crescita delle funzioni: l'importanza dei logaritmi

*Logarithms arise in any process where things are repeatedly halved. Ex.:*

- **Trees**
  - *What is the height h of a tree with d children and n leaf nodes?*
- **Binary Search**
- **Bits:** *How many bits do we need to represent any of n possibilities?*
- **Multiplication:**  $a^b = \exp(\ln(a^b)) = \exp(b \ln a)$
- **Fast exponentiation**
  - $a^n = (a^{n/2})^2$  if  $a$  is even
  - $a^n = a(a^{[n/2]})^2$  if  $a$  is odd
  - This requires  $O(\lg n)$  mult. =>
- **Summation**
  - Harmonic numbers:  $H(n) = \sum_{i=1}^n \frac{1}{i} \sim \ln n$



```
function power(a,n)
    if (n = 0) return(1)
    x = power(a, [n/2])
    if (n is even) then return(x2)
                    else return(a * x2)
```

# Crescita delle funzioni: l'importanza dei logaritmi



S&T:

**Importance of an Even Split**

**Problem:** *How many queries does binary search take on the million-name Manhattan phone book if each split was 1/3 to 2/3 instead of 1/2 to 1/2?*

$$\log_{3/2}(1,000,000) \approx 35 \text{ chiamate nel caso peggiore}$$
$$\log_2(1,000,000) \approx 20 \text{ chiamate nel caso peggiore}$$

## Take-home lesson

*The power of binary search comes from its logarithmic complexity, not the base of the log.*