DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

La complessità computazionale

Cinzia Pizzi

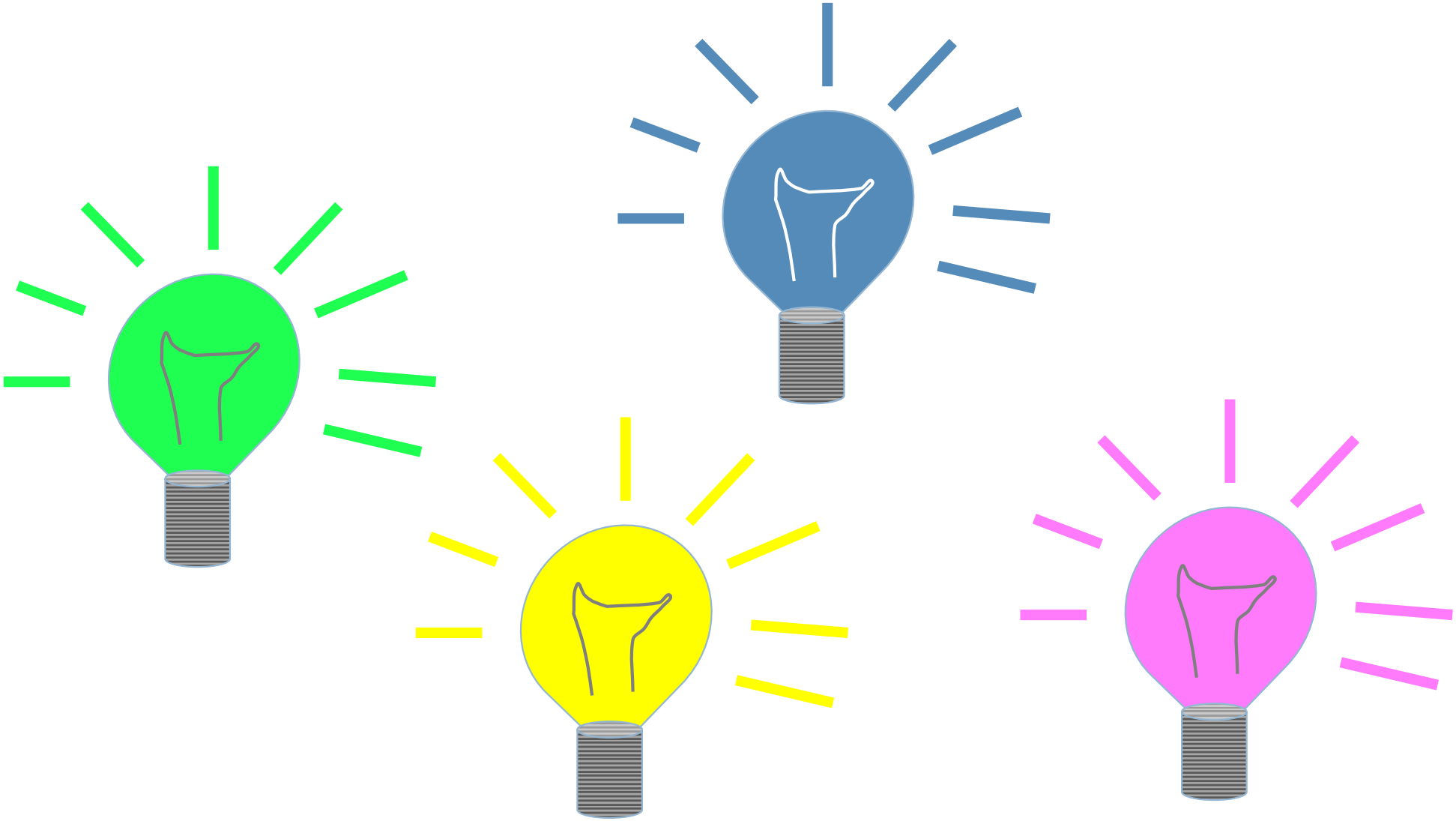
Obiettivi

- Conoscenza dei principali metodi di misura delle prestazioni degli algoritmi
- Capacità di ricavare la complessità computazionale di semplici algoritmi e comprenderne il significato
- Capacità di confrontare e classificare gli algoritmi in base alla loro complessità

CODICE

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Un problema , tante soluzioni



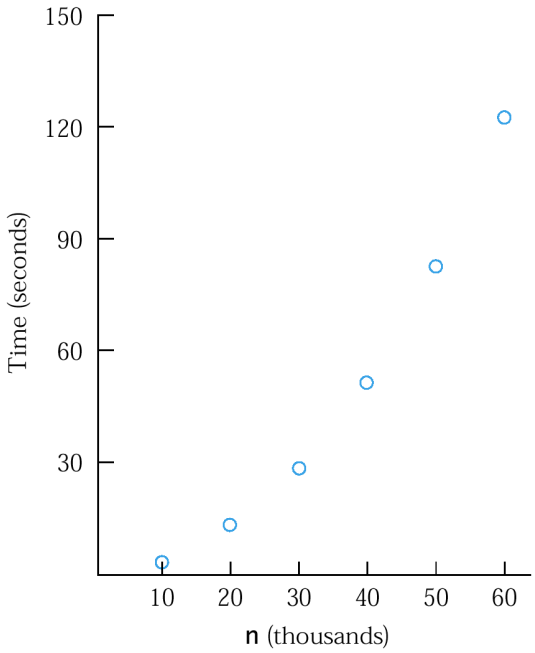
Analisi delle prestazioni

- Tipi di analisi degli algoritmi
  - Sperimentale
  - Teorica
- Cosa si misura
  - Tempo di esecuzione
  - Memoria occupata
- Obiettivi: ricavare informazioni per
  - Fare previsioni sulle performance di un algoritmo
  - Confrontare diverse soluzioni algoritmiche

Analisi sperimentale

Rilevare le prestazioni

- Per valutare l'efficienza temporale di un algoritmo si misura il tempo impiegato per la sua esecuzione su insiemi di dati di dimensioni diverse, per poi magari fare un grafico



Rilevare le prestazioni

- Il tempo ***non va misurato con un cronometro***, perché alcune componenti del tempo reale di esecuzione non dipendono dall'algoritmo stesso
  - caricamento in memoria della JVM da parte del S.O.
  - caricamento delle classi del programma
  - lettura dei dati dallo standard input
  - visualizzazione dei risultati
- Tali componenti sono, poi, anche variabili nel tempo, da un'esecuzione all'altra (bisognerebbe prendere un tempo di esecuzione medio), per effetto di altri programmi in esecuzione sul computer

Rilevare le prestazioni

- **Il tempo di esecuzione di un algoritmo va misurato all'interno del programma**

- Si può usare il metodo statico **System.currentTimeMillis()**

  che, a ogni invocazione, restituisce un valore di tipo **long** che rappresenta…
  - il numero di millisecondi trascorsi da un evento di riferimento (***la mezzanotte del 1 gennaio 1970***) !!

- Ciò che interessa è la *differenza* tra due valori
  - si invoca **System.currentTimeMillis( )** *prima e dopo* l'esecuzione dell'algoritmo (escludendo le operazioni di input/output dei dati)

Rilevare le prestazioni

```
public static void main(String[] args)
{  double[] a = ...;
   // fase di input dei dati
   ...
   long initTime = System.currentTimeMillis();

   invocazione al metodo che implementa l'algoritmo
   di cui si vogliono misurare le prestazioni

   long finalTime = System.currentTimeMillis();
   // calcolo i secondi trascorsi
   long elapsedTime =
         Math.round((finalTime - initTime) / 1000.0);
   System.out.print("Tempo di esecuzione "+ elapsedTime);
   System.out.println(" secondi");
   // fase di output dei dati
   ...
}
```

Esempio

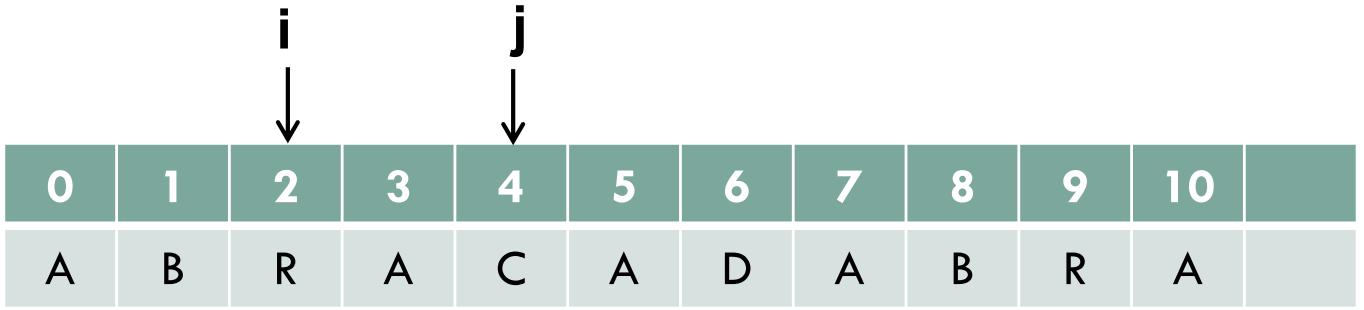
- Consideriamo il seguente codice, dato n:

```
count ← 0
for i ← 0 to n-1
        for j ← i to n-1
                scrivi (i,j)
                count  ← count +1
```

- Utilizza due indici
  - i che varia da 0 a n -1
  - j che varia da i a n -1

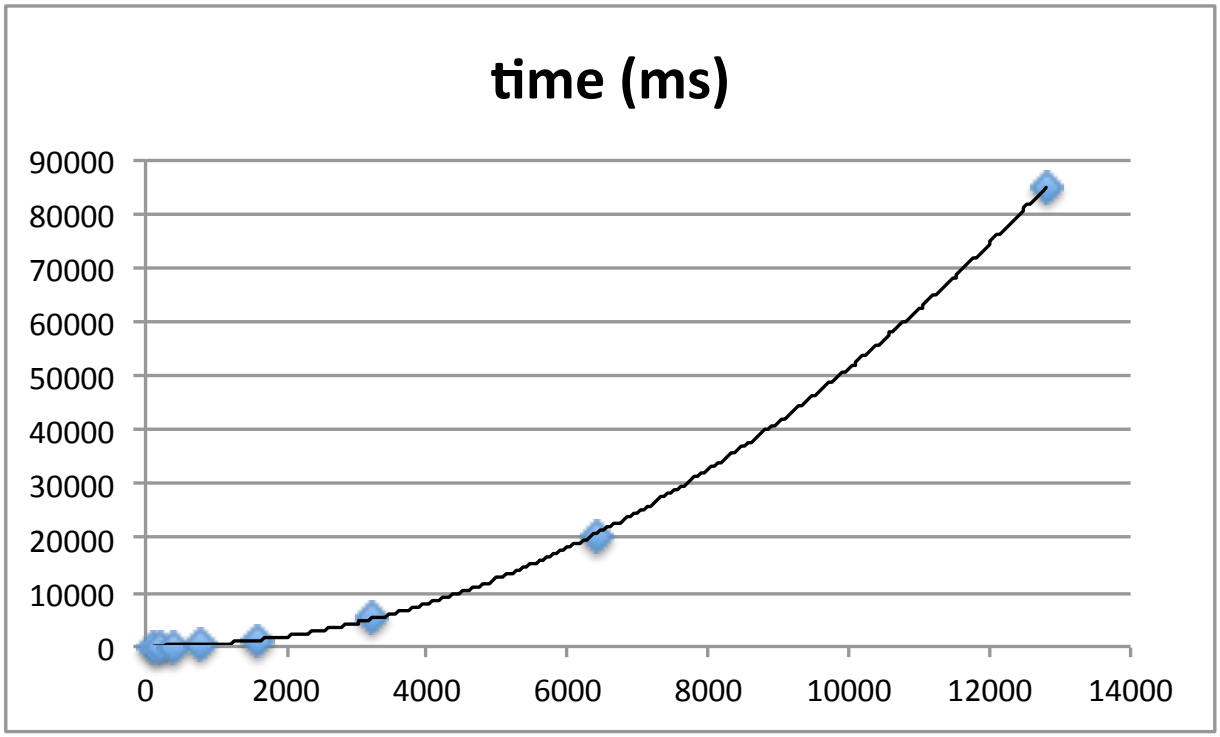
Possibile applicazione

- Data una stringa di caratteri di lunghezza n voglio sapere qual è il numero di sottostringhe presenti



- Oppure: elencare tutte le sottostringhe di una stringa
  - Le coppie di indici rappresentano le posizioni di inizio e fine di ciascuna sottostringa
  - Attenzione: se volessi stampare anche le sottostringhe dovrei tener conto anche delle operazioni di estrazione della sottostringa

Misuriamo i tempi di esecuzione al crescere di n



Che andamento ha la curva?

- L'andamento della curva:
  - Non è lineare
    - $T = a\ n$ sarebbe una retta
  - Non è esponenziale
    - $T = a^n$ salirebbe molto più in fretta
  - E' polinomiale, ovvero del tipo $a\ n^b$
    - Ma quanto vale b? 2? 3? 2.5?

Uno sguardo ai dati

input	time	ratio	log ratio
100	5	-	-
200	21	4.20	2.07
400	80	3.81	1.93
800	321	4.01	2.00
1600	1279	3.98	1.99
3200	5279	4.13	2.05
6400	20551	3.89	1.96
12800	84872	4.13	2.05

Un po' di conti…

- **Assumendo** che l'andamento sia del tipo **a n^b**
  - Ricaviamo **b** dal log-ratio tra valori consecutivi quando raddoppiamo la taglia dell'input
    - Nel nostro caso b=2
- Perché?
  - T(k-1) e T(k) i tempi di due rilevazioni successive per input n e 2n, rispettivamente:
    - $T(k) = [a\ (2n)^b]$ e $T(k-1) = [a\ n^b]$
  - $T(k)/T(k-1) = [a\ (2n)^b] / [a\ n^b] = [a\ 2^b\ n^b]/[a\ n^b] = 2^b$
  - $b = \log_2 [T(k)/T(k-1)]$

Un po' di conti…

- **Assumendo** che l'andamento sia del tipo **a n^b**
  - Ricaviamo **a** prendendo un valore abbastanza grande di n, per esempio 6400 e risolviamo $T = a\ n^b$
    - $20551 = a \times 6400^2 \Rightarrow a = 5 \times 10^{-4}$
  - Proviamo a fare previsioni per n=12800
    - $T = 5 \times 10^{-4} \times 12800^2 = 82204$ ms = 82 sec
    - Tempo realmente misurato 84 sec !!!

Considerazioni/1

- In realtà i tempi misurati sperimentalmente possono ancora essere soggetti ad elementi esterni
  - il contenuto dell'input
  - le caratteristiche hardware
  - il linguaggio di programmazione
  - il compilatore…

Considerazioni/2

- Possiamo confrontare due algoritmi a parità di condizioni sperimentali…
  - Importante: ripetere l'esecuzione più volte e fare la media dei tempi misurati
- …ma non saremo completamente indipendenti dalla macchina

Analisi teorica

Analisi teorica di un algoritmo

- Basata su tipo e frequenza delle operazioni
- Analisi di tipo qualitativo
- Permette
  - Di avere informazioni sulle prestazioni senza implementare l'algoritmo!
  - Confronto tra algoritmi
  - Classificazione di algoritmi in classi di complessità

Analisi teorica delle prestazioni

- Il tempo d'esecuzione di un algoritmo dipende dal numero e dal tipo di istruzioni in ***linguaggio macchina*** eseguite dal processore

- Per fare un'analisi teorica senza realizzare un algoritmo, compilarlo e tradurlo in linguaggio macchina dobbiamo fare delle ***semplificazioni drastiche***

Modello di costo

- Assumiamo che il tempo di esecuzione dipenda da
  - Numero di operazioni primitive (passi base)
  - Dimensione dei dati da elaborare
  - Valore dei dati stessi

Passi base

- Operazioni che hanno tempo di esecuzione
  - Costante (circa)
  - Indipendente dal numero, tipo e valore dei dati
- Assegnazione di valore ad una variabile
- Un'operazione aritmetica
- Un'operazione logica o di confronto
- Un accesso in lettura/scrittura ad una cella di un array
- **Un enunciato contenente una invocazione di un metodo NON è un'operazione primitiva**

Dimensione dell'input

- Si indica con la lettera n
- Significato dipende dal problema
  - Grandezza di un numero (problemi di calcolo)
  - Numero di elementi da ordinare (problemi di ordinamento)
  - …
- Indipendentemente dal tipo di dati, indichiamo sempre con ***n*** la dimensione dell'input

Valore dei dati

- In alcuni problemi il valore dei dati influenza le prestazioni
  - Es. Ordinamento di una sequenza
- Analisi teorica comprende
  - Caso peggiore
    - Limite superiore alle prestazioni
  - Caso medio
    - Indicazione sulle prestazioni attese
  - Caso migliore
    - Casi fortunati

Notazioni asintotiche

- Conteggio della frequenza dei passi base => espressioni complesse
- Andamento qualitativo => notazione semplificata
  - Ordine di grandezza della complessità dell'algoritmo
- Il limite superiore asintotico all'andamento di una funzione è detta O-grande

$$f(n) \in O(g(n)) \quad se \quad \exists c > 0 \; e \; n_0 > 0 \quad tali \; che \quad f(n) \leq cg(n) \quad \forall n \geq n_0$$

Notazioni asintotiche

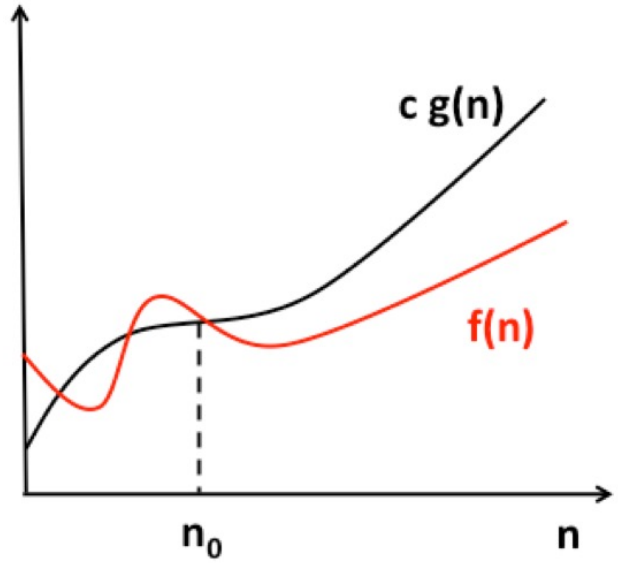
- Limite inferiore asintotico per una funzione f è detto Omega-grande

$$f(n) \in \Omega(g(n)) \;\; se \quad \exists c > 0 \; e \; n_0 > 0 \quad tali \; che \quad f(n) \geq cg(n) \quad \forall n \geq n_0$$

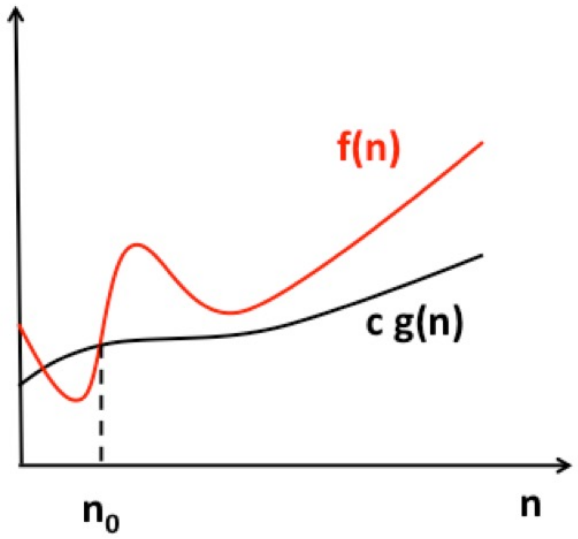
- Notazione Theta-grande

$$f(n) \in \theta(g(n)) \;\; se \quad f(n) \in O(g(n)) \;\; e \; f(n) \in \Omega(g(n))$$

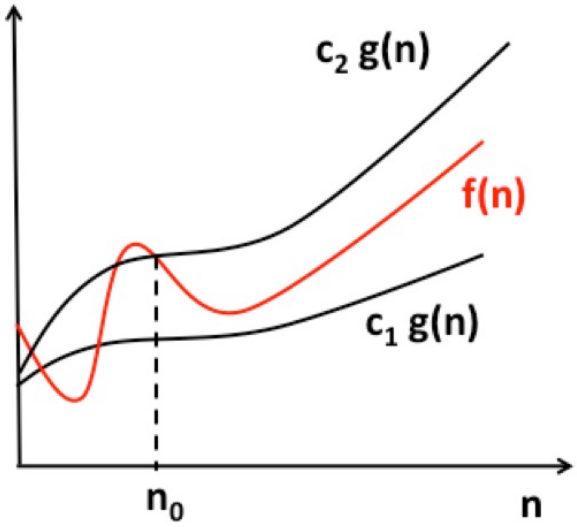
Notazioni asintotiche



$f(n) \in O(g(n))$



$f(n) \in \Omega(g(n))$



$f(n) \in \Theta(g(n))$

Come ottenere O-grande

- A partire dall'espressione che conta i passi base in termini di taglia dell'input n
  - Si considera il termine dominante
  - Si ignorano le costanti moltiplicative
- **Si dimostra che: Una funzione polinomiale è O-grande del suo monomio di grado massimo, con coefficiente moltiplicativo arbitrario**
  - **Quindi, per semplicità, tale coefficiente viene assunto uguale a 1**

Come ottenere O-grande

- In termini crescente di complessità

$$O(1) \subset O(logn) \subset O(n) \subset O(nlogn) \subset O(n^2) \subset O(n^3) \subset 2^n \subset n^n$$

- Esempi

$$6 \in O(1)$$

$$5 + 3\log n + n^2 \in O(n^2)$$

$$5n + 3n^3 + 10n^2 \in O(n^3)$$

$$100n + 5n^{15} + 3 \cdot 2^n \in O(2^n)$$

Complessità di un programma strutturato

- Dati due blocchi di istruzioni B1 e B2
  - Se sono in sequenza o alternativi (IF-THEN-ELSE)
    - La complessità totale è il massimo tra le complessità di B1 e B2
    - Max(O(B1), O(B2))
  - Se sono due blocchi annidati
    - La complessità totale è il prodotto delle due complessità
    - O(B1) x O(B2)

Esempio 1 complessita' costante

- Calcolare la complessita' di:

```
n ← 0
n ← n+2
stampa(n)
```

- Tre istruzioni:
  - inizializzazione della variabile n
  - incremento di una quantità pari a 2
  - stampa a video del valore (si intende stampa di un solo numero)
- Totale passi base = 3. Poiché nella notazione asintotica O-grande le costanti moltiplicative non vengono evidenziate, otteniamo **O(1)**

Esempio 1bis complessita' metodo

```
n ← 0          } B1
n ← n+2        }
stampa(n)      } B2
```

- Calcolare la complessità di:
- Tre istruzioni:
  - inizializzazione della variabile n
  - incremento di una quantità pari a 2
  - Invocazione metodo di stampa
- Gli assegnamenti contribuiscono con un certo numero costante di passi base e posso vederli come un blocco B1
- La chiamata a metodo posso vederla come un blocco B2
- La complessità totale sarà il massimo tra le complessità dei due blocchi

Esempio 1bis complessita' metodo

- Ipotizziamo quattro diverse implementazioni di stampa(n)

1)
```
public static void stampa(int n){
    System.out.println("Il numero e'"+n);
}
```

- Tempo costante: faccio una sola operazione, indipendentemente dal valore n dell'input

- La complessità sia di B1 che di B2 è costante, per cui lo sarà anche la complessità totale

Esempio 1bis complessita' metodo

- Ipotizziamo quattro diverse implementazioni di stampa(n)

2)

```
public static void stampa(int n){
    for(int i=0; i<n; i++)
        System.out.println("Il numero e'"+i);
}
```

- Il metodo effettua un numero lineare di passi base (n operazioni di stampa di un numero)
- La complessità di B1e' costante mentre la complessità di B2 è lineare. La complessità totale è il massimo delle due, quindi lineare, ovvero O(n).

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Esempio 1bis complessita' metodo

- Ipotizziamo quattro diverse implementazioni di stampa(n)

3)

```
public static void stampa(int n){
    for(int i=0; i<n²; i++)
        System.out.println("Il numero e'"+i);
}
```

- Il metodo effettua un numero quadratico di passi base rispetto a n (n^2 operazioni di stampa di un numero)
- La complessità di B1 e' costante mentre la complessità di B2 è quadratica. La complessità totale è il massimo delle due, quindi quadratica, ovvero $O(n^2)$.

Esempio 1bis complessita' metodo

- Ipotizziamo quattro diverse implementazioni di stampa(n)

4)

```
public static void stampa(int n){
    for(int i=0; i<n²; i+=n)
        System.out.println("Il numero e'"+i);
}
```

- Il metodo effettua un numero lineare di passi base (stampa di un numero) rispetto a n (i deve arrivare a n^2 ma ad ogni ciclo aumenta di n)
- La complessità di B1 e' costante mentre la complessità di B2 è lineare. La complessità totale è il massimo delle due, quindi lineare, ovvero O(n).

Esempio 2 complessita' lineare

```
i ← 0
while (i<n) do
      i ← i+1
stampa(i)
```

- Sia n il valore di input, calcolare complessità del codice
- Istruzioni
  - inizializzazione della variabile i
  - Valutazione di (i<n) effettuata n+1 volte
  - n volte viene eseguito l'incremento di i
  - n volte viene eseguito l'assegnamento a i
  - Stampa di un valore
- Totale passi base $1 + (n+1) + 2n + 1 = 3n+3$
- Complessità asintotica **O(n)**

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Esempio 3 complessita' quadratica

```
val ← 1
for i ← 1 to n
      scrivi val
      for j ← 1 to n
            val ← val x 2
            val ← val +1
scrivi val
```

- Sia n il valore di input, calcolare complessità del codice
- Istruzioni
  - inizializzazione della variabile val
  - Complessità cicli annidati: il ciclo for esterno viene eseguito n volte, così come il ciclo interno
  - Stampa variabile val
- Totale passi base $1+ n \times (1+ n \times 2) + 1 = 2n^2 + n + 2$
- Complessità asintotica $\mathbf{O(n^2)}$

Esempio 4 complessita' quadratica

```
for i ← 1 to n
    fun(i)
```

- Sia n il valore di input, fun una funzione che esegue un numero di passi base pari al valore dell'argomento passato come parametro. Calcolare la complessità del codice
- Istruzioni
  - Il ciclo for viene eseguito n volte: ad ogni iterazione i invoca la funzione fun(i) che conterà per i passi base

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{n}{2}$$

Formula di Gauss

- Semplificando otteniamo **$O(n^2)$**

Formula di Gauss

- $1 + 2 + 3 + \ldots + n = n(n+1)/2$

$$
\begin{array}{l}
1 + 2 + 3 + 4 + \ldots + n/2 + \\
n + n\text{-}1 + n\text{-}2 + n\text{-}3 + \ldots + n/2+1 = \\
\hline
n+1 + n+1 + n+1 + n+1 + \ldots + n+1 =
\end{array}
$$

$$= (n+1)\, n/2$$

Complessita' logaritmica

```
count ← 0
i ← 1
while( i < n){
      count++
      i = i*2
}
```

- Supponiamo di raddoppiare l'indice i che si muove nell'intervallo da 1 a n
- i = 1; 2; 4; 8; …
- count conta il numero di iterazioni
- Dopo k iterazioni, count = k e i = 2^k
- Quando si ferma il ciclo?
  - i = n => 2^k = n => k = $\log_2 n$
  - Il ciclo si ferma dopo $\log_2 n$ iterazioni, ad ogni iterazione compie un numero di operazioni costante
  - Complessita' totale $O(\log_2 n)$

Complessita' logaritmica

- Vediamo un altro esempio

$$\log_{10}(n^{22}) = 22 \log_{10} n \in O(\log_{10} n)$$

- Ricordiamo, però, che

$$\log_a n = (1/\log_c a) \log_c n$$

- Cioè il cambio di base in un logaritmo equivale all'applicazione di un coefficiente moltiplicativo, quindi in un'espressione O-grande si può omettere l'indicazione della base del logaritmo, così come si può omettere l'esponente a cui sia eventualmente elevato l'argomento del logaritmo stesso

$$\forall a > 1, \forall c \neq 0: \log_a n^c \in O(\log_a n) \equiv O(\log n)$$

Proprieta' notazione O-grande

- La definizione matematica implica che
  - se $f(n) \in O(g(n))$, allora è anche $f(n) \in O(h(n))$ per ogni funzione **h(n)** che cresca più velocemente di **g(n)**

- Nota: la caratterizzazione che interessa di più è quella più precisa, cioè ***più stringente*** o "nei minimi termini"

- Però se, ad esempio, $T(n) \in O(n^2)$, **non è sbagliato** (ma è poco utile…) dire che

  $T(n) \in O(n^3)$ oppure $T(n) \in O(n^4)$ perché

  $T(n) \in O(n^2) \subset O(n^3) \subset O(n^4) \subset \ldots$

Classi di complessità

- Negli esempi abbiamo ottenuto sempre espressioni che descrivono complessità polinomiali
- Attenzione! Non tutti i problemi hanno una soluzione algoritmica polinomiale
- Esistono diverse classi di complessità in informatica
  - Esistono problemi intrinsecamente "difficili", per cui non è ancora stata trovata una soluzione che non sia (almeno) esponenziale

Classi di complessità

- In informatica esistono due grandi classi di complessità computazionale
- La classe P
  - Problemi risolvibili al più in tempo polinomiale ("facili")
- La classe NP
  - Problemi per cui non si conosce (ancora) una soluzione polinomiale ("difficili") e che richiedono tempo (almeno) esponenziale

Esempi

- Data una sequenza di n interi con segno…
- 1-SUM: trovare gli elementi con valore 0
  - 1 ciclo: O(n)
- 2-SUM: trovare le coppie di elementi la cui somma è 0
  - 2 cicli: $O(n^2)$
- 3-SUM: trovare le triplette di elementi la cui somma è 0
  - 3 cicli: $O(n^3)$
- SUBSET-SUM: esiste un sottoinsieme la cui somma è 0 ?
  - NP (naive $n2^n$)

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

P = NP? 1000 000 $ Prize

E in pratica come uso O-grande?

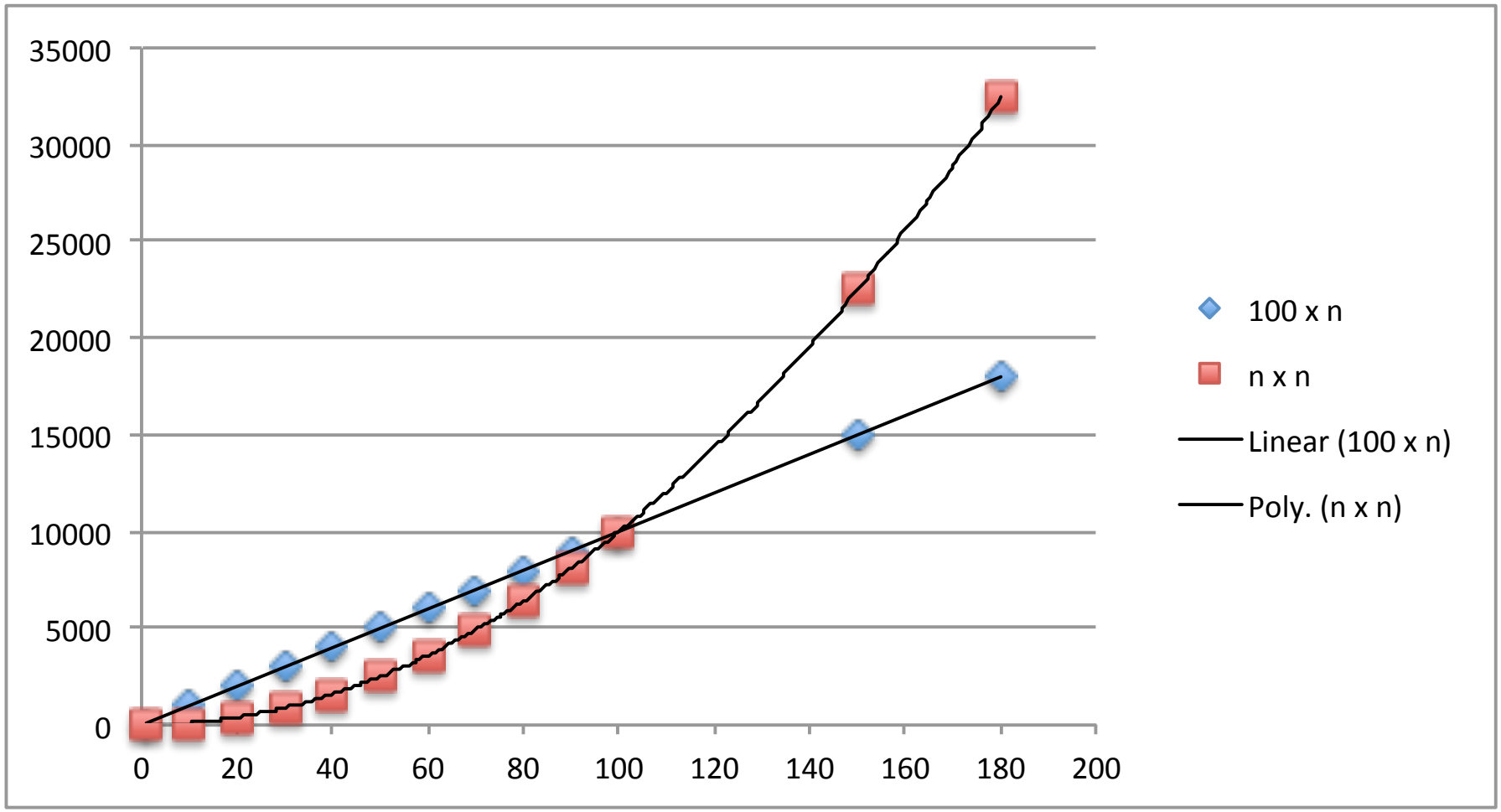
- Si osservi che la complessità di due algoritmi può essere in entrambi i casi O(g(n)) anche se il secondo è, per esempio, mille volte più veloce del primo
  - $T_1(n) = 10000\ n$ e $T_2(n) = 10\ n$ sono entrambi O(n)

- La notazione O-grande è utile per determinare l'applicabilità di principio di un algoritmo

- Sia l'analisi teorica e l'analisi sperimentale sono importanti

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Lato pratico: valutare sempre il range dei dati da



Esercizio

- Data una funzione di complessità quadratica $O(n^2)$, se per n = 10 il tempo di esecuzione è T=30sec, per n' = 20, qual è il tempo di esecuzione atteso?

- 400 sec
- 60 sec
- 120 sec
- Più di 1000 sec.

Ragionamento

- Complessita' quadratica $O(n^2)$, tempo T
- n' = 20 = 2 x n
- $(n')^2 = (2n)^2 = 4\ n^2$
- Se per n utilizzando un algoritmo quadratico ci metto T, per n' ci metto 4T
- T= 30sec => T' = 120sec

Ragionamento

- Complessita' cubica $O(n^3)$, tempo T=30s
- n' = 20 = 2 x n
- $(n')^3 = (2n)^3 = 8\ n^3$
- Se per n utilizzando un algoritmo cubico ci metto T, per n' ci metto 8T
- T= 30sec => T' = 240sec

Ragionamento

- Complessita' cubica $O(n^3)$, tempo T=20sec con input n
- n' = 20 = 2 x n
- $(n')^3 = (2n)^3 = 2^3 n^3 = 8 n^3$
- Se per n utilizzando un algoritmo cubico ci metto T, per n' ci metto 8T
- T= 20sec => T' = 160sec

Take home message

- L'analisi delle prestazioni di un algoritmo può essere
  - Sperimentale
  - Teorica
- Entrambe sono importanti e ci danno informazioni sulla bontà di un algoritmo

- Permettono di
  - Predire i tempi di esecuzione degli algoritmi
  - Confrontare algoritmi
  - Classificare algoritmi