# Analisi degli algoritmi

Progettazione di Algoritmi a.a. 2018-19

Matricole congrue a 1

Docente: Annalisa De Bonis



## Analisi degli algoritmi

- E' possibile progettare diversi algoritmi per risolvere uno stesso problema
  - Si pensi ad esempio agli algoritmi di ordinamento di n numeri: Merge Sort, Quick Sort, Insertion Sort, Bubble Sort, Selection Sort, Heap Sort, ...
- Un algoritmo può impiegare molto meno tempo di un altro
  - MergeSort: tempo proporzionale a nlog n
  - QuickSort: tempo nel caso peggiore proporzionale a $n^2$

  o usare molto meno spazio di un altro
  - Alcuni algoritmi di ordinamento non utilizzano strutture dati ausiliarie in quanto ordinano "sul posto" andando a modificare la posizione degli elementi all'interno della sequenza input. Questi algorimi richiedono solo una piccola quantità di memoria aggiuntiva che è di molto interiore inferiore alla dimensione n dell'input.
    - Esempi: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort,....

## Analisi degli algoritmi

- è utile avere un modo per confrontare tra loro diverse soluzioni per capire quale sia la migliore. Migliore in base ad un certo criterio di efficienza, come ad esempio uso della memoria o velocità .
- Abbiamo bisogno di tecniche di analisi che consentano di valutare un algoritmo solo in base alle sue caratteristiche e non quelle del codice che lo implementa o della macchina su cui è eseguito.
- Come informatici, oltre a dover essere in grado di trovare soluzioni ai problemi, dobbiamo essere in grado di valutare la nostra soluzione e capire se c'è margine di miglioramento.
  - Limiti inferiori



## Efficienza degli algoritmi

Proviamo a definire la nozione di efficienza (rispetto al tempo di esecuzione):

- ***Un algoritmo è efficiente se, quando è implementato, viene eseguito velocemente su istanze input reali.***
  - Concetto molto vago.
    - Non chiarisce **dove** viene eseguito l'algoritmo e **quanto veloce** deve essere la sua esecuzione
      - Anche un algoritmo molto cattivo può essere eseguito molto velocemente se è applicato a un input molto piccolo o se è eseguito con un processore molto veloce
      - Anche un algoritmo molto buono può richiedere molto tempo per essere eseguito se implementato male
    - …

## Efficienza degli algoritmi

- ...
- Non chiarisce cosa è un'istanza input reale
  - Noi non conosciamo a priori tutte le possibili istanze input reali
  - Alcune istanze potrebbero essere più "cattive" di altre
- Inoltre non fa capire come la velocità di esecuzione dell'algoritmo deve variare al crescere della dimensione dell'input
- Due algoritmi possono avere tempi di esecuzione simili per input piccoli ma tempi di esecuzione molto diversi per input grandi



## Efficienza degli algoritmi

- Vogliamo una definizione concreta di efficienza che
  - sia indipendente dal processore
  - indipendente dal tipo di istanza
  - dia una misura di come aumenta il tempo di esecuzione al crescere della dimensione dell'input.

## Efficienza

Forza bruta. Per molti problemi non triviali, esiste un naturale algoritmo di forza bruta che controlla ogni possibile soluzione.

- Tipicamente impiega tempo $2^N$ (o peggio) per input di dimensione N.
- Non accettabile in pratica.

- Esempio:
  - Voglio ordinare in modo crescente un array di N numeri distinti
  - Soluzione (ingenua) esponenziale: permuto i numeri ogni volta in modo diverso fino a che ottengo la permutazione ordinata (posso verificare se una permutazione è ordinata con al più N-1 confronti, confrontando ciascun elemento con il successivo)
    - Nel caso pessimo genero N! permutazioni
    - NB: $N! > 2^N$ per n>3



## Efficienza

- **Problemi con l'approccio basato sulla ricerca esaustiva nello spazio di tutte le possibili soluzioni (forza bruta)**
  - **Ovviamente richiede molto tempo**
  - **Non fornisce alcuna informazione sulla struttura del problema che vogliamo risolvere.**

- **Proviamo a ridefinire la nozione di efficienza:** Un algoritmo è efficiente se ha una performance migliore, da un punto di vista analitico, dell'algoritmo di forza bruta.

- **Definizione molto utile**. Algoritmi che hanno performance migliori rispetto agli algoritmi di forza bruta di solito usano euristiche interessanti e forniscono informazioni rilevanti sulla struttura intrinseca del problema e sulla sua trattabilità computazionale.
- **Problema con questa definizione**. Anche questa definizione è vaga. Cosa vuol dire "perfomance migliore"?

## Tempo polinomiale

**Proprietà desiderata.** Quando la dimensione dell'input raddoppia, l'algoritmo dovrebbe risultare più lento solo di un fattore costante c

**Mergesort:**

per N --> tempo c x N x log N;

per 2N --> tempo c x 2N x log(2N)= 2 x c x N x (logN +1)
= 2 x c x N x logN + 2 x c x N
≤ 2 x c x N x logN + 2 x c x N x log N
= **4 x** c x N x logN per ogni N>1;

aumenta di al più **4** volte

**Algoritmo di forza bruta:** per N tempo c x (N-1) x N!

Per 2N tempo c x (2N-1) x (2N)! =c x (2N-1) x (2N x (2N-1) x ⋯ x (N+1) x N!)
> c x 2 x (N-1) x N! X N! = **(2xN!)**x c x (N-1) xN!

(ultima disuguaglianza perchè (2N-1) > 2x(N-1) e 2N x (2N-1) x ⋯ x (N+1) > N! )

2xN! non è una costante



## Tempo polinomiale

Def. Si dice che un algoritmo impiega tempo polinomiale (poly-time) se quando la dimensione dell'input raddoppia, l'algoritmo risulta più lento solo di un fattore costante c

Esistono due costanti c > 0 e d > 0 tali che su ciascun input di dimensione N, il numero di passi è limitato da $c\,N^d$ .

Se si passa da un input di dimensione N ad uno di dimensione 2N allora il tempo di esecuzione passa da $c\,N^d$ a $c\,(2N)^d = c2^dN^d$
NB: $2^d$ è una costante

## Analisi del caso pessimo

Tempo di esecuzione nel caso pessimo. Ottenere un bound sul **più grande tempo di esecuzione possibile** per tutti gli input di una certa dimensione N.

- In genere è una buona misura di come si comportano gli algoritmi nella pratica
- Approccio "pessimistico" (in molti casi l'algoritmo potrebbe comportarsi molto meglio)
  - ma è difficile trovare un'alternativa efficace a questo approccio

Tempo di esecuzione nel caso medio. Ottenere un bound al tempo di esecuzione su un **input random** in funzione di una certa dimensione N dell'input.

- Difficile se non impossibile modellare in modo accurato istanze reali del problema mediante distribuzioni input.
- Un algoritmo disegnato per una certa distribuzione di probabilità sull'input potrebbe comportarsi molto male in presenza di altre distribuzioni.



## Tempo polinomiale nel caso pessimo

Def. Un algoritmo è **efficiente** se il suo tempo di esecuzione nel caso pessimo è polinomiale.

Motivazione: Funziona veramente in pratica!

- Sebbene $6.02 \times 10^{23} \times N^{20}$ sia, da un punto di vista tecnico, polinomiale, un algoritmo che impiega questo tempo potrebbe essere inutile in pratica.
- Per fortuna, i problemi per cui esistono algoritmi che li risolvono in tempo polinomiale, quasi sempre ammettono algoritmi polinomiali il cui tempo di esecuzione è proporzionale a polinomi che crescono in modo moderato ($c \times N^d$ con c e d piccoli).
- Progettare un algoritmo polinomiale porta a scoprire importanti informazioni sulla struttura del problema

## Eccezioni

- Alcuni algoritmi polinomiali hanno costanti e/o esponenti grandi e sono inutili nella pratica
- Alcuni algoritmi esponenziali sono largamente usati perchè il caso pessimo si presenta molto raramente.
  - Esempio: algoritmo del simplesso per risolvere problemi di programmazione lineare



## Perchè l'analisi della complessità è importante

La tabella riporta i tempi di esecuzione su input di dimensione crescente, per un processore che esegue un milione di istruzioni per secondo.
Nei casi in cui il tempo di esecuzione è maggiore di $10^{25}$ anni, la tabella indica che il tempo richiesto è molto lungo (very long)

| | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $1.5^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 4 sec |
| $n = 30$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 18 min | $10^{25}$ years |
| $n = 50$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 11 min | 36 years | very long |
| $n = 100$ | < 1 sec | < 1 sec | < 1 sec | 1 sec | 12,892 years | $10^{17}$ years | very long |
| $n = 1{,}000$ | < 1 sec | < 1 sec | 1 sec | 18 min | very long | very long | very long |
| $n = 10{,}000$ | < 1 sec | < 1 sec | 2 min | 12 days | very long | very long | very long |
| $n = 100{,}000$ | < 1 sec | 2 sec | 3 hours | 32 years | very long | very long | very long |
| $n = 1{,}000{,}000$ | 1 sec | 20 sec | 12 days | 31,710 years | very long | very long | very long |

Analisi degli algoritmi

Esempio:

```
InsertionSort(a):        //n è la lunghezza di a
 For(i=1;i<n;i=i+1){
   elemDaIns=a[i];
   j=i-1;
   While((j≧0)&& a[j]>elemDaIns){ //cerca il posto per a[i]
        a[j+1]=a[j];      //shifto a destra gli elementi più grandi
        j=j-1;
}
 a[j+1]=elemDaIns;
}
```



# Analisi di InsertionSort

$t_i$ è il numero di iterazioni del ciclo di while all'i-esima iterazione del for

| InsertionSort(a): | Costo | Num. Volte |
|---|---|---|
| For(i= 1;i <n;i=i+1){ | $c_1$ | n |
| elemDaIns=a[i]; | $c_2$ | n-1 |
| j=i-1; | $c_3$ | n-1 |
| While((j ≧0)&& a[j]>elemDaIns){ | $c_4$ | $\sum_{i=1}^{n-1} t_i$ |
| a[j+1]=a[j]; | $c_5$ | $\sum_{i=1}^{n-1} (t_i-1)$ |
| j=j-1; | $c_6$ | $\sum_{i=1}^{n-1} (t_i-1)$ |
| } | | |
| a[j+1]=elemDaIns; | $c_7$ | n-1 |
| } | | |

## Analisi di InsertionSort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n} (t_i - 1) + c_7(n-1)$$

Nel caso pessimo $t_i$ =i+1 per ogni i (elementi in ordine decrescente)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} (i+1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n} i + c_7(n-1)$$



## Analisi di InsertionSort

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} (i+1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n-1} i + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4((\sum_{i=1}^{n-1} i) + n - 1) + c_5 \sum_{i=1}^{n-1} i + c_6 \sum_{i=1}^{n-1} i + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{(n-1)n}{2} + n - 1) + c_5(\frac{(n-1)n}{2}) + c_6(\frac{(n-1)n}{2}) + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n^2}{2} + \frac{n}{2} - 1) + c_5(\frac{n^2}{2} - \frac{n}{2}) + c_6(\frac{n^2}{2} - \frac{n}{2}) + c_7(n-1) \\
&= (c_4 + c_5 + c_6)\frac{n^2}{2} + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 - c_3 - c_4 - c_7) \\
&= an^2 + bn + c
\end{aligned}$$

## Ordine di grandezza

- Nell'analizzare la complessità di InsertionSort abbiamo operato delle astrazioni
- Abbiamo ignorato il valore esatto prima delle costanti $c_i$ e poi delle costanti a, b e c.
- Il calcolo di queste costanti per alcuni algoritmi può essere molto stancante ed è inutile rispetto alla classificazione degli algoritmi che vogliamo ottenere.
- Queste costanti inoltre dipendono
  - dalla macchina su cui si esegue il programma
  - dal tipo di operazioni che contiamo
    - Operazioni del linguaggio ad alto livello
    - Istruzioni di basso livello in linguaggio macchina



## Ordine di grandezza

- Possiamo aumentare il livello di astrazione considerando solo l'ordine di grandezza
- Consideriamo solo il termine "dominante"
  - Per InsertionSort: $an^2$
  - Giustificazione: più **grande** è n, minore è il contributo dato dagli altri termini alla stima della complessità
- Ignoriamo del tutto le costanti
  - Diremo che il tempo di esecuzione di InsertionSort ha ordine di grandezza $n^2$
  - Giustificazione: più grande è n, minore è il contributo dato dalle costanti alla stima della complessità

# Efficienza asintotica degli algoritmi

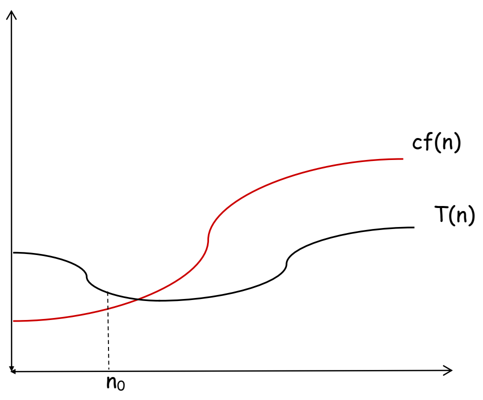
- Per input piccoli può non essere corretto considerare solo l'ordine di grandezza ma per input "abbastanza" grandi è corretto farlo

- Esempio: $10n^2+100n+10$
  per $n<10$, il secondo termine è maggiore del primo
  man mano che n cresce il contributo dato dai termini meno significativi diminuisce



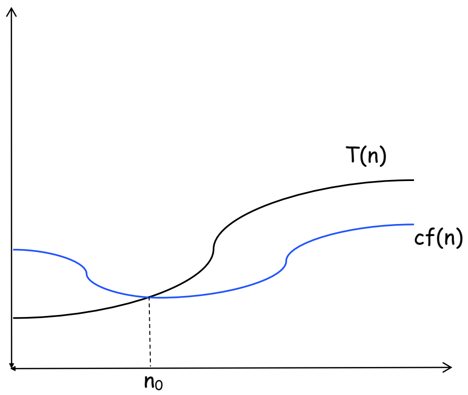
## Ordine asintotico di grandezza

Limiti superiori. $T(n)$ è $O(f(n))$ se esistono delle costanti $c > 0$ ed $n_0 \geq 0$ tali che per tutti gli $n \geq n_0$ si ha $T(n) \leq c \cdot f(n)$.



## Ordine asintotico di grandezza

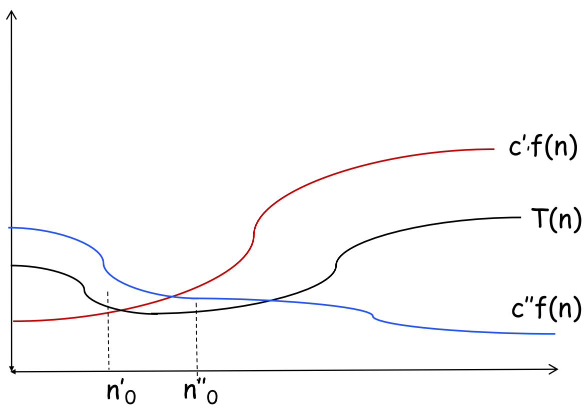
**Limiti inferiori.** T(n) è $\Omega(f(n))$ se esistono costanti $c > 0$ ed $n_0 \geq 0$ tali che per tutti gli $n \geq n_0$ si ha $T(n) \geq c \cdot f(n)$.






## Ordine asintotico di grandezza

**Limiti esatti.** T(n) è $\Theta(f(n))$ se T(n) sia $O(f(n))$ che $\Omega(f(n))$.



## Ordine asintotico di grandezza

- Quando analizziamo un algoritmo miriamo a trovare stime asintotiche quanto più "strette" è possibile
- Dire che InsertionSort ha tempo di esecuzione $O(n^3)$ non è errato ma $O(n^3)$ non è un limite "stretto" in quanto si può dimostrare che InsertionSort ha tempo di esecuzione $O(n^2)$
- $O(n^2)$ è un limite stretto?
  - Sì, perché il numero di passi eseguiti da InsertionSort è $an^2+bn+c$, con $a>0$, che non solo è $O(n^2)$ ma è anche $\Omega(n^2)$.
  - Si può dire quindi che il tempo di esecuzione di InsertionSort è $\Theta(n^2)$



## Errore comune

Affermazione priva di senso. Ogni algoritmo basato sui confronti richiede almeno $O(n \log n)$ confronti.

- **Per i lower bound si usa $\Omega$**

Affermazione corretta. Ogni algoritmo basato sui confronti richiede almeno $\Omega(n \log n)$ confronti.

## Proprietà

Transitività .
- Se $f = O(g)$ e $g = O(h)$ allora $f = O(h)$.
- Se $f = \Omega(g)$ e $g = \Omega(h)$ allora $f = \Omega(h)$.
- Se $f = \Theta(g)$ e $g = \Theta(h)$ allora $f = \Theta(h)$.

Additività .
- Se $f = O(h)$ e $g = O(h)$ allora $f + g = O(h)$.
- Se $f = \Omega(h)$ e $g = \Omega(h)$ allora $f + g = \Omega(h)$.
- Se $f = \Theta(h)$ e $g = \Theta(h)$ allora $f + g = \Theta(h)$.



## Bound asintotici per alcune funzioni di uso comune

Polinomi. $a_0 + a_1 n + \ldots + a_d n^d$ , con $a_d > 0$ , è $\Theta(n^d)$.

Dim. $O(n^d)$: Basta prendere $n_0=1$ e come costante c la somma $(|a_0|+ |a_1| + \ldots +|a_d|)$

Infatti

$a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$

$\leq |a_0|+ |a_1|n + |a_2|n^2 + \ldots +|a_d|n^d$

$\leq (|a_0|+ |a_1|+ |a_2| + \ldots +|a_d|)\, n^d$, per ogni $n \geq 1$.

## Bound asintotici per alcune funzioni di uso comune

Polinomi.

$a_0 + a_1n + \ldots + a_dn^d$ , con $a_d > 0$, è $\Theta(n^d)$.

Dimostriamo come esercizio che $a_0 + a_1n + \ldots + a_dn^d$ è anche $\Omega(n^d)$:

- $a_0 + a_1n + \ldots + a_dn^d \geqq a_dn^d -(|a_0|+ |a_1|n + \ldots +|a_{d-1}|n^{d-1})$
- Abbiamo appena visto che un polinomio di grado d è $O(n^d)$
  - Ciò implica $a_0+ |a_1|n + \ldots +|a_{d-1}|n^{d-1} = O(n^{d-1})$

    e di conseguenza esistono $n'_0 \geq 0$ e $c' > 0$

    tali che $a_0+ |a_1|n + \ldots +|a_{d-1}|n^{d-1} \leq c'n^{d-1}$ per ogni $n \geq n'_0$

- Quindi $a_dn^d -(a_0+ |a_1|n + \ldots +|a_{d-1}|n^{d-1}) \geq a_dn^d - c'n^{d-1}$ per ogni $n \geq n'_0$



## Bound asintotici per alcune funzioni di uso comune

- Per dimostrare $a_0 + a_1n + \ldots + a_dn^d = \Omega(n^d)$ dobbiamo trovare le costanti $n_0 \geq 0$ e $c>0$ tali che $a_0 + a_1n + \ldots + a_dn^d \geq cn^d$ per ogni $n \geq n_0$
- Nella slide precedente abbiamo dimostrato che esistono due costanti $n'_0 \geq 0$ e $c' > 0$ tali che $a_dn^d -(a_0+ |a_1|n + \ldots +|a_{d-1}|n^{d-1}) \geq a_dn^d - c'n^{d-1}$ per ogni $n \geq n'_0$ .
- Quindi per dimostrare $a_0 + a_1n + \ldots + a_dn^d = \Omega(n^d)$ è sufficiente trovare due costanti $n_0 \geq 0$ e $c>0$ tali che $a_dn^d - c'n^{d-1} \geq cn^d$ per ogni $n \geq n_0$
- Risolvendo la disequazione $a_dn^d - c'n^{d-1} \geq cn^d$ si ha $c \leq a_d - c'/n$ .
- Prendiamo allora $c = a_d - c'/n$. Siccome deve essere $c>0$, imponiamo $a_d - c'/n > 0$ che è soddisfatta per $n > c'/a$. Prendiamo allora $n_0 = \max\{n_0', 2c'/a_d\}$

In conclusione abbiamo $n_0 = \max\{n_0', 2c'/a_d\}$ e $c = a_d - c'/n$.

## Ordine asintotico di grandezza

Esempio:

$T(n) = 32n^2 + 17n + 32$.

- $T(n)$ è $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$ e $\Theta(n^2)$ .
- $T(n)$ non è $O(n)$, $\Omega(n^3)$, $\Theta(n)$ o $\Theta(n^3)$.



## Tempo lineare: O(n)

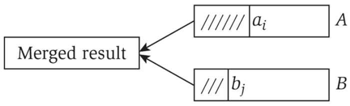
Tempo lineare. Il tempo di esecuzione è al più un fattore costante per la dimensione dell'input.

Esempio:
Computazione del massimo. Computa il massimo di n numeri $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
    Se (ai > max)
        max ← ai
}
```

## Tempo lineare: O(n)

Merge. Combinare 2 liste ordinate $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_m$ in una lista ordinata.




```
i = 1, j = 1
while (i≤n and j≤m) {
   if (aᵢ ≤ bⱼ) aggiungi aᵢ alla fine della lista output e incrementa i
   else aggiungi bⱼ alla fine della lista output e incrementa j
}
Aggiungi alla lista output gli elementi non ancora esaminati di una
delle due liste input
```

Affermazione. Fondere liste di dimensione n richiede tempo O(n+m).
Dim. Dopo ogni confronto, la lunghezza dell'output aumenta di 1.



## Tempo quadratico: $O(n^2)$

Tempo quadratico. Tipicamente si ha quando un algoritmo esamina tutte le coppie di elementi input

Coppia di punti più vicina. Data una lista di n punti del piano $(x_1, y_1), \ldots, (x_n, y_n)$, vogliamo trovare la coppia più vicina.

Soluzione $O(n^2)$. Calcola la distanza tra tutte le coppie di punti.

```
min ← (x₁ - x₂)² + (y₁ - y₂)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xᵢ - xⱼ)² + (yᵢ - yⱼ)²
      Se (d < min)
         min ← d
   }
}
```

Per effettuare i confronti non c'è bisogno di estrarre la radice quadrata

NB. è possibile fare meglio

## Cubic Time: $O(n^3)$

Tempo cubico. Tipicamente si ha quando un algoritmo esamina tutte le triple di elementi.

Disgiunzione di insiemi. Dati n insiemi $S_1, \ldots, S_n$ ciascuno dei quali è un sottoinsieme di {1, 2, …, n}, c'è qualche coppia di insiemi che è disgiunta?
Soluzione $O(n^3)$. Per ogni coppia di insiemi, determinare se i due insiemi sono disgiunti. (Supponiamo di poter determinare in tempo costante se un elemento appartiene ad un insieme)

```
flag = true
for i = 1 to n{      //corpo iterato n volte
 for j = i+1 to n { //corpo iterato n-i volte ad ogni iterazione del for esterno
      foreach elemento p di Si { //corpo iterato al più n volte ad ogni iteraz. for su j
         if p appartiene anche a Sj //supponiamo test richiede ogni volta  O(1)
             flag = false; break;
      }
   If(flag = true)// nessun elemento di Si appartiene a Sj
          riporta che Si  e  Sj sono disgiunti
  }
}
```



## Un'utile richiamo

Alcune utili proprietà dei logaritmi:

1. $\log_a x = (\log_b x) / (\log_b a)$
2. $\log_a(xy) = \log_a x + \log_a y$
3. $\log_a x^k = k \log_a x$

Dalla 1. discende:

4. $\log_a x = 1/(\log_x a)$

Dalla 3. discende:

5. $\log_a(1/x) = -\log_a x$

Dalla 2. e dalla 5 discende:

6. $\log_a(x/y) = \log_a x - \log_a y$

## Bound asintotici per alcune funzioni di uso comune

Logaritmi. $O(\log_a n) = O(\log_b n)$ per ogni costante a, b > 0.

Logaritmi. $\log n = O(n)$.

Dim. Dimostriamo per induzione che $\log_2 n \le n$ per ogni $n \ge 1$.

Base dell'induzione: Vero per n=1.

Passo Induttivo: Supponiamo $\log_2 n \le n$ vera per n.

Dimostriamo che è vera per n+1.

1. $\log_2(n+1) \le \log_2 (2n) = \log_2 2 + \log_2 n = 1 + \log_2 n$

Per ipotesi induttiva $\log_2 n \le n$ e quindi

2. $1+\log_2 n \le n+1$.

Dalla catena di disuguaglianze 1. e dalla disuguaglianza 2. si ha

$\log_2(n+1) \le n+1$.



## Un utile richiamo

Parte intera inferiore:

La parte intera inferiore di un numero x è denotata con $\lfloor x \rfloor$ ed è definita come quell'unico intero per cui vale che $x-1 < \lfloor x \rfloor \le x$. In altre parole, $\lfloor x \rfloor$ è il più grande intero minore o uguale di x.

Esempio: $\lfloor 4.3 \rfloor=4$ , $\lfloor 6.9 \rfloor=6$ , $\lfloor 3 \rfloor=3$

Proprietà 1: L'intero più piccolo strettamente maggiore di x è $\lfloor x \rfloor+1$.

Dim. Dalla def. di $\lfloor x \rfloor$ si ha $x-1< \lfloor x \rfloor \le x$ . La prima disequazione implica $x < \lfloor x \rfloor+1$ .

Le disequazioni $x < \lfloor x \rfloor+ 1$ e $\lfloor x \rfloor \le x$ implicano la proprietà .

Proprietà 2: $\lfloor \lfloor a/b \rfloor/c \rfloor = \lfloor a/(bc) \rfloor$ , per a, b e c interi con b e c diversi da 0

## Un utile richiamo

Parte intera superiore:
La parte intera superiore di un numero x è denotata con ⌈x⌉ ed è definita come quell'unico intero per cui vale che $x \leq \lceil x \rceil < x+1$ In altre parole, ⌈x⌉ è il più piccolo intero maggiore o uguale di x.
Esempio: ⌈4.3⌉ =5 , ⌈6.9⌉ =7 , ⌈3⌉=3

Proprietà 3: L'intero più grande strettamente minore di x è ⌈x⌉ -1.
Dim. Dalla def. di ⌈x⌉ si ha $x \leq \lceil x \rceil < x+1$ . La seconda disequazione implica $\lceil x \rceil - 1 < x$.
Le disequazioni $x \leq \lceil x \rceil$ e $\lceil x \rceil - 1 < x$ implicano la proprietà .

Proprietà 4: $\lceil \lceil a/b \rceil / \rceil = \lceil a/(bc) \rceil$ per a, b e c interi con b e c diversi da 0



## Tempo logaritmico

Tipicamente si ha quando ogni passo riduce di un fattore costante il numero di passi che restano da fare

```
For (i=1; i<= n;i=i*2)
  print(i)
```

Il for in alto richiede tempo $\Theta(\log n)$

**Dimostrazione** : Il for termina quando i diventa maggiore di n.

- Ad ogni iterazione il valore di i raddoppia → dopo la k-esima iterazione $i = 2^k$.
- Per sapere dopo quante iterazioni termina il for dobbiamo trovare **il più piccolo k per cui $2^k > n$.** In altre parole vogliamo k tale che $2^k > n$ **e** $2^{k-1} \leq n$
- Risolvendo le disequazioni $2^k > n$ e $2^{k-1} \leq n$ otteniamo $2^k > n \leftrightarrow k > \log_2 n$ e $2^{k-1} \leq n \leftrightarrow k-1 \leq \log_2 n$
- Le due disuguaglianze ottenute implicano $\log_2 n - 1 < k - 1 \leq \log_2 n$ e quindi si ha $k-1 = \lfloor \log_2 n \rfloor$ da cui $k = \lfloor \log_2 n \rfloor + 1$ (Potevamo usare direttamente la Proprietà 1 per determinare k) .
- Dopo esattamente $k = \lfloor \log_2 n \rfloor + 1$ iterazioni $i=2^k$ diventa più grande di n. → Numero iterazioni è $\lfloor \log_2 n \rfloor + 1 = \Theta(\log n)$

**N.B.** Se invece di raddoppiare, il valore di i viene moltiplicato per una generica costante c>1 allora la base del log è c ma ai fini della valutazione asintotica non cambia niente.

## Tempo logaritmico

Per esercizio dimostriamo che anche il seguente for richiede tempo $O(\log n)$

```
For (i=n; i ≧ 1;i=⌊i/2⌋)
  print(i)
```

**Dimostrazione : Qui dimostriamo O(logn).**
Il for termina quando i diventa minore di 1.
Ad ogni iterazione il valore di i è minore o uguale della metà del valore che aveva in precedenza → dopo la k-esima iterazione $i \leqq n/2^k$ (il valore esatto è $i=\lfloor n/2^k \rfloor$).

Per sapere dopo quante iterazioni termina il for dobbiamo trovare il più piccolo k per cui $n/2^k < 1$ .

$$n/2^k < 1 \leftrightarrow 2^k > n \leftrightarrow k > \log_2 n.$$

$\lfloor \log_2 n \rfloor + 1 > \log_2 n$ per cui dopo al più $k=\lfloor \log n \rfloor +1$ iterazioni sicuramente $n/2^k$ è più piccolo di 1 → max numero iterazioni $\leqq \lfloor \log n \rfloor +1 =O(\log n)$



## Tempo logaritmico

Qui dimostriamo che il for della slide precedente richiede tempo il $\Omega(\log n)$
**Dimostrazione :**
Per dimostrare il tempo $\Omega(\log n)$ ragioniamo sul valore esatto di i dopo m iterazioni. Tale valore è $\lfloor n/2^m \rfloor$ (si dimostra con la Proprietà 2).

Dobbiamo determinare per quali valori di m risulta $\lfloor n/2^m \rfloor \geqq 1$ cioè i valori m per cui si ha che il ciclo **non** si arresta dopo m iterazioni.

Siccome $\lfloor n/2^m \rfloor > n/2^m -1$ allora se m è tale che $n/2^m - 1 \geqq 0$ allora $\lfloor n/2^m \rfloor \geqq 1$ .
Risolviamo allora $n/2^m -1 \geqq 0$

$$n/2^m -1 \geqq 0 \leftrightarrow n/2^m \geqq 1 \leftrightarrow n \geqq 2^m \leftrightarrow m \leqq \log n \leftrightarrow m \leqq \lfloor \log n \rfloor$$

Quindi fino a che il numero di iterazioni m è minore o uguale di $\lfloor \log n \rfloor$, il ciclo di for non termina.
Deduciamo allora che il numero totale di iterazioni è maggiore o uguale di $\lfloor \log n \rfloor+1$. Questo basta a dimostrare $\Omega(\log n)$.
Se vogliamo una stima esatta del numero di iterazioni, osserviamo che nella slide precedente abbiamo dimostrato che il numero di iterazioni è almeno $\lfloor \log n \rfloor +1$ Ne conseque che il numero esatto di iterazioni è $\lfloor \log n \rfloor +1$.

## Tempo logaritmico: O(log n)

Tipicamente si ha quando ogni passo riduce di un fattore costante il numero di passi che restano da fare

Ricerca binaria. Dato un array A ordinato di n numeri ed un numero x vogliamo determinare se x è in A

```
binarySearch(A, ,n, x)
 l = 0;
 r =n
while l <= r
     c= (l+r)/ 2 //assumiamo troncamento
     if x = A[c]
          return true
     if x < A[c]
          r =c-1
     else l=c+1 //caso x>A[c]
return false
```

Se la dimensione r-l+1 dell'intervallo [l,r] è pari allora il sottointervallo di destra [c+1,r] ha un elemento in più rispetto a quello di sinistra. In caso contrario i due sottointervalli hanno la stessa dimensione.
Caso r-l+1 pari: intervallo di sinistra ha $\lfloor (r-l+1)/2 \rfloor - 1$ elementi e quello di destra $\lfloor (r-l+1)/2 \rfloor$.
Caso r-l+1 dispari: entrambi gli intervalli hanno $\lfloor (r-l+1)/2 \rfloor$ elementi



## Tempo logaritmico: O(log n)

Analisi ricerca binaria.

Il while termina quando l>r, cioè quanto il range [l,r] vuoto.

- Inizialmente [l,r]=[0,n-1] e quindi contiene n elementi
- Dopo la prima iterazione, [l,r] contiene al più $\lfloor n/2 \rfloor$ elementi
- Dopo la seconda iterazione, [l,r] contiene al più $\lfloor n/4 \rfloor$ elementi
- Dopo la terza iterazione, [l,r] contiene al più $\lfloor n/8 \rfloor$ elementi
- ...
- Dopo la i-esima iterazione, [l,r] contiene al più $\lfloor n/2^i \rfloor$ elementi
- Per sapere quando termina il ciclo di while dobbiamo trovare il più piccolo i per cui $\lfloor n/2^i \rfloor < 1$
- Poiché $n/2^i < 1$ implica anche $\lfloor n/2^i \rfloor < 1$ allora calcoliamo il più piccolo i tale che $n/2^i < 1$
- Risolviamo la disequazione $n/2^i < 1$
  - $n/2^i < 1 \leftrightarrow n < 2^i \leftrightarrow \log n < i$
- Quindi se prendiamo **i uguale al più piccolo intero maggiore di log n** allora otteniamo che $\lfloor n/2^i \rfloor < 1$ e di conseguenza sappiamo che, per quel valore di i, il ciclo termina all'i-esima iterazione. Tale intero i è $\lfloor \log n \rfloor + 1$ .

## Analisi ricerca binaria per ottenere Θ(log n)

Poichè stiamo esaminando il caso pessimo, assumiamo che ad ogni iterazione del while la ricerca continui nel sottointervallo di [l,r] più grande. All'i-esima iterazione, il più grande dei due sottointervalli di [l,r] ha lunghezza pari esattamente a $\lfloor n/2^i \rfloor$.

Il while termina quando l>r, cioè quanto il range [l,r] vuoto.

- Per trovare una stima esatta del numero di iterazioni del ciclo di while dobbiamo trovare l'intero i per cui $\lfloor n/2^i \rfloor < 1$ e $\lfloor n/2^{i-1} \rfloor \geqq 1$ . Risolviamole entrambe:

1. Abbiamo già visto nella slide precedente che $\lfloor n/2^i \rfloor < 1$ per $i > \log n$
2. Risolviamo la disequazione $\lfloor n/2^{i-1} \rfloor \geqq 1$ .

   Dalla def. di parte intera inf. si ha $\lfloor n/2^{i-1} \rfloor > n/2^{i-1} - 1$ per cui se $n/2^{i-1} - 1 \geqq 0$ allora $\lfloor n/2^{i-1} \rfloor > 1$.

   Risolviamo allora $n/2^{i-1} - 1 \geqq 0$

   - $n/2^{i-1} - 1 \geqq 0 \leftrightarrow n/2^{i-1} \geqq 1 \leftrightarrow n \geqq 2^{i-1} \leftrightarrow \log n \geqq i-1$
   - Quindi mettendo insieme la 1 e la 2 abbiamo $\log n - 1 < i-1 \leqq \log n$. Per def. di parte intera inf. $i-1 = \lfloor \log n \rfloor$ e quindi $i = \lfloor \log n \rfloor + 1$ .



| Espressione $O$ | nome |
|---|---|
| $O(1)$ | costante |
| $O(\log \log n)$ | $\log \log$ |
| $O(\log n)$ | logaritmico |
| $O(\sqrt[c]{n}),\ c > 1$ | sublineare |
| $O(n)$ | lineare |
| $O(n \log n)$ | $n \log n$ |
| $O(n^2)$ | quadratico |
| $O(n^3)$ | cubico |
| $O(n^k)\ (k \geq 1)$ | polinomiale |
| $O(a^n)\ (a > 1)$ | esponenziale |

## Tempo O(√n)

```
j=0;
i=0;
while(i<=n){
   j++;
   i=i+j;
}
```

**Analisi :**
Il while termina quando i diventa maggiore di n.
All'iterazione k al valore di i viene sommato j=k per cui dopo aver iterato il while k volte il valore di i è (1+2+3+…+k) =k(k+1)/2.
Per sapere quante iterazioni vengono fatte dobbiamo calcolare il più piccolo k tale che k(k+1)/2 >n . Per semplicità osserviamo che $k^2/2 \leq k(k+1)/2$ per cui se $k^2/2 > n$ allora k(k+1)/2 > n . Risolviamo $k^2/2 > n$ .

$$k^2/2 > n \leftrightarrow k^2 > 2n \leftrightarrow k > (2n)^{1/2}$$

Dalla proprietà 1 il più piccolo intero k maggiore di > $(2n)^{1/2}$ è $\lfloor(2n)^{1/2}\rfloor+1 = O(\sqrt{n})$



## Regole per la notazione asintotica

$d(n) = O(f(n)) \Rightarrow ad(n) = O(f(n)),\ \forall$ costante $a > 0$
Es.: $\log n = O(n) \Rightarrow 7\log n = O(n)$

$d(n) = O(f(n)), e(n) = O(g(n)) \Rightarrow d(n)+e(n) = O(f(n)+g(n))$
Es.: $\log n = O(n), \sqrt{n} = O(n) \Rightarrow \log n + \sqrt{n} = O(n)$

$d(n) = O(f(n)), e(n) = O(g(n)) \Rightarrow d(n)e(n) = O(f(n)g(n))$
Es.: $\log n = O(\sqrt{n}), \sqrt{n} = O(\sqrt{n}) \Rightarrow \sqrt{n}\log n = O(n)$

$d(n) = O(f(n)), f(n) = O(g(n)) \Rightarrow d(n) = O(g(n))$
Es.: $\log n = O(\sqrt{n}), \sqrt{n} = O(n) \Rightarrow \log n = O(n)$

$f(n) = a_d n^d + \cdots a_1 n + a_0 \Rightarrow f(n) = O(n^d)$
Es.: $5n^7 + 6n^4 + 3n^3 + 100 = O(n^7)$

$n^x = O(a^n),\ \forall$ costanti $x > 0, a > 1$ Es.: $n^{100} = O(2^n)$

## Regole per la notazione asintotica

- Le prime 5 regole nella slide precedente valgono anche se sostituiamo O con Ω o con Θ
- La quarta regola è chiamata proprietà transitiva



## Logaritmi a confronto con polinomi e radici

Per ogni **costante** $x > 0$, $\log n = O(n^x)$. (N.B. x può essere < 1)

Dim. **Se $x \geq 1$** si ha $n \leq n^x$ per ogni $n \geq 0$ e quindi $n=O(n^x)$. Abbiamo già dimostrato che $\log n=O(n)$ per cui dalla proprietà transitiva si ha $\log n=O(n^x)$

**Consideriamo il caso $x<1$**. Vogliamo trovare le costanti $c>0$ e $n_0 \geq 0$ tale che $\log n \leq cn^x$ per ogni $n \geq n_0$

Siccome sappiamo che $\log_2 m < m$ per ogni $m \geq 1$ allora ponendo $m=n^x$ con $n \geq 1$, si ha $\log_2 n^x < n^x$ da cui $x\log_2 n < n^x$ e dividendo entrambi i membri per x si ha $\log_2 n < 1/x\ n^x$. Perchè la disequazione $\log_2 n \leq cn^x$ sia soddisfatta, basta quindi prendere $\mathbf{c=1/x}$ e $\mathbf{n_0=1}$.

**NB: nella notazione asintotica possiamo eliminare la base del log**

## Potenze di logaritmi a confronto con polinomi e radici

Per ogni $x > 0$ e $b>0$ **costanti**, $(\log n)^b = O(n^x)$.

Dim.

Vogliamo trovare le costanti $c>0$ e $n_0 \geq 0$ tali che $(\log n)^b \leq cn^x$ per ogni $n \geq n_0$

Risolviamo la disequazione $(\log n)^b \leq cn^x$ :

$(\log n)^b \leq cn^x \leftrightarrow \log n \leq (cn^x)^{1/b} = c^{1/b} n^{x/b}$ (se e solo se vale perchè $\log n>0$)

Troviamo le costanti $c>0$ ed $n_0 \geq 0$ tali che $\log n \leq c^{1/b}n^{x/b}$ per ogni $n \geq n_0$

Abbiamo già dimostrato nella slide precedente che $\log n=O(n^y)$ per ogni $y>0$. Ciò vale anche se poniamo $y= x/b$. Quindi esistono due costanti $c'>0$ e $n'_0 \geq 0$ tali che $\log n \leq c'n^{x/b}$ per ogni $n \geq n'_0$.

Di conseguenza basta imporre $c^{1/b} = c'$ ed $n_0 = n'_0$ da cui $c=(c')^b$ ed $n_0 = n'_0$



## Potenze di logaritmi a confronto con polinomi e radici

Dimostrare che per ogni $x > 0$ , $a>0$ e $b>0$ **costanti** , $(\log n^a)^b = O(n^x)$.

La dimostrazione è molto semplice se si usa quanto visto nelle slide precedenti

## Tempo O(n log n)

Tempo O(n log n). Viene fuori quando si esamina la complessità di algoritmi basati sulla tecnica del divide et impera

Ordinamento. Mergesort e heapsort sono algoritmi di ordinamento che effettuano O(n log n) confronti.

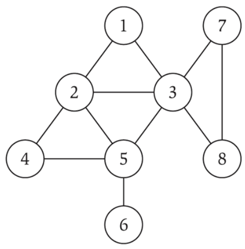
Il più grande intervallo vuoto. Dati n time-stamp $x_1, \ldots, x_n$ che indicano gli istanti in cui le copie di un file arrivano al server, vogliamo determinare qual è l'intervallo di tempo più grande in cui non arriva alcuna copia del file.

Soluzione O(n log n). Ordina in modo non decrescente i time stamp. Scandisci la lista ordinata dall'inizio computando la differenza tra ciascun istante e quello successivo. Prendi il massimo delle differenza calcolate. Tempo O(nlog n+n)=O(nlogn)



## Grafo

- Esempio (vedremo meglio questo concetto nelle prossime lezioni)




V = { 1, 2, 3, 4, 5, 6, 7, 8 }
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }
n = 8
m = 11

## Tempo polinomiale $O(n^k)$

Insieme indipendente di dimensione k (k costante). Dato un grafo, esistono k nodi tali che nessuno coppia di nodi è connessa da un arco?

Soluzione $O(n^k)$. Enumerare tutti i sottoinsiemi di k nodi.

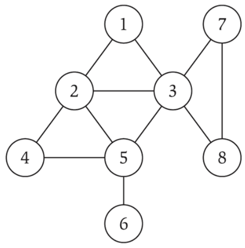
```
foreach sottoinsieme S di k nodi {
   controlla se S è un insieme indipendente
   if (S è un insieme indipendente)
      riporta che S è in insieme indipendente
   }
}
```

- Controllare se S è un insieme indipendente = $O(k^2)$
- Numero di sottoinsiemi di k elementi = $\binom{n}{k} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{k(k-1)(k-2)\ldots(2)(1)} \le \frac{n^k}{k!}$
- Tempo totale $O(k^2\, n^k / k!) = O(n^k)$



## Insieme indipendente

- Esempio: per k=3 l'algoritmo riporta gli insiemi {1,4,6}, {1,4,7}, {1,4,8},{1,5,7},{1,5,8},{1,6,7}, {1,6,8}, {2,6,7},{2,6,8},{3,4,6}, {4,6,7},{4,6,8}




V = { 1, 2, 3, 4, 5, 6, 7, 8 }
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }
n = 8
m = 11

## Tempo esponenziale

Esempio:
Massimo insieme indipendente . Dato un grafo G, qual è la dimensione massima di un insieme indipendente di G?

Def. insieme indipendente: un insieme indipendente di un grafo è un sottoinsieme di vertici a due a due non adiacenti?

Soluzione $O(n^2 2^n)$. Esamina tutti i sottoinsiemi di vertici.
NB: Il numero totale di sottoinsiemi di un insieme di n elementi è $2^n$

```
S* ← ϕ
foreach sottoinsieme S of nodi {
   controlla se S è un insieme indipendente
   Se (S è il più grande  insieme indipendente visto finora)
      aggiorna S* ← S
   }
}
```



## Tempo esponenziale

Per esercizio proviamo che il tempo del nostro algoritmo per il massimo insieme indipendente è $\Theta(n^2 2^n)$

Dim: Assumiamo per semplicità n dispari

- Stimiamo il tempo per controllare l'indipendenza degli insiemi di dimensione maggiore o uguale di $\lceil n/2 \rceil$.

1. il tempo per controllare l'indipendenza di ciascuno di questi insiemi è almeno $\Omega(n^2)$ perché dobbiamo controllare almeno $n/2(n/2-1)/2$ coppie di nodi nel caso pessimo.
2. Il numero di insiemi di dimensione maggiore o uguale di $\lceil n/2 \rceil$ è $2^{n-1}$ in quanto per ogni insieme di dimensione k , per $k=\lceil n/2 \rceil,\ldots,n$, ve ne è esattamente uno di dimensione n-k. Quindi se divido gli insiemi di dimensione al più $\lceil n/2 \rceil-1$ da quelli di dimensione maggiore o uguale di $\lceil n/2 \rceil$, divido i $2^n$ insiemi in due metà uguali.

- 1. e 2. → Tempo totale per controllare insiemi di dimensione maggiore o uguale di $\lceil n/2 \rceil$ è almeno $\Omega(n^2 2^{n-1}) = \Omega(n^2 2^n/2) = \Omega(n^2 2^n)$ → Algoritmo ha tempo $\Omega(n^2 2^n)$

Siccome vale sia $O(n^2 2^n)$ (slide precedente) che $\Omega(n^2 2^n)$ allora abbiamo dimostrato che il tempo è $\Theta(n^2 2^n)$ . E se n è pari?

Esercizio:

Ci chiediamo: $3^n$ è $O(2^n)$?

Sappiamo che $3^n$ è $O(2^n)$ se e solo se esistono due costanti $c>0$ ed $n_0 \geq 0$ t.c. $3^n \leq c \cdot (2^n)$ per ogni $n \geq n_0$.

Proviamo a determinare tali costanti risolvendo la disequazione $3^n \leq c \cdot 2^n$ rispetto a c

$3^n \leq c \cdot 2^n \leftrightarrow c \geq 3^n/2^n=(3/2)^n$. Occorre quindi prendere $c \geq (3/2)^n$ .

La funzione $(3/2)^n$ cresce al crescere di n e tende all'infinito al tendere di n all'infinito. Siccome $(3/2)^n$ tende all'infinito, qualsiasi valore scegliamo per la costante c, questo valore sarà superato da $(3/2)^n$ per n sufficientemente grande.

Ne deduciamo che **non** esistono $c>0$ ed $n_0 \geq 0$ t.c. $3^n \leq c \cdot (2^n)$ per ogni $n \geq n_0$.

→ Abbiamo dimostrato che $3^n$ non è $O(2^n)$