Analisi degli algoritmi

Progettazione di Algoritmi a.a. 2017-18

Matricole congrue a 1

Docente: Annalisa De Bonis

Efficienza degli algoritmi

- **Proviamo a definire la nozione di efficienza**: ***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 input molto piccolo ed è eseguito con un processore molto veloce
    - Anche un algoritmo molto buono può richiedere molto tempo per essere eseguito se implementato male
  - Non chiarisce cosa è un'istanza input reale
    - Noi non conosciamo a priori tutte le possibili istanze input
  - 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,
  - e che 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 in tempo O(N) confrontando ciascun elemento con il successivo)
    - Nel caso pessimo genero N! permutazioni
    - NB: $N! = N \times (N-1) \times (N-2) \ldots \times 1 > 2^N$ per ogni $N > 3$

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,1 l'algoritmo dovrebbe risultare 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 tempo 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

Algoritmo di ordinamento efficiente (mergesort): per N tempo cxNxlog N;

per 2N tempo cx2Nx log(2N)=2c x Nx(logN +1)

<2c x N x logN+2cxN

≤**4**c x N x logN, per ogni N>1;

Il tempo aumenta di **4** volte al raddoppiare dell'input

Algoritmo di ordinamento di forza bruta: per N tempo c xN!

Per 2N tempo cx(2N)! > c x **(2**x**N!)** x N! 2xN! non è una costante

Analisi del caso pessimo

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

- In genere è una buona misura dell'efficienza degli 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 $6.02 \times 10^{23} \times N^{20}$ è da un punto di vista tecnico polinomiale, un algoritmo che impiega questo tempo potrebbe essere inutile in pratica.
- In pratica, 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

N.B. Si stima che il pianeta Terra abbia circa 4,54 x10^9 anni...

Analisi degli algoritmi

Esempio:

```
 InsertionSort(a):        //n e` la lunghezza di a
  For(i=2;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 piu` 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=2;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	$\sum_{i=2}^{n} t_i$
a[j+1]=a[j];	c_5	$\sum_{i=2}^{n} (t_i-1)$
j=j-1;	c_6	$\sum_{i=2}^{n} (t_i-1)$
}		
a[j+1]=elemDaIns;	c_7	n-1
}		

Analisi di InsertionSort

$$T(n)=c_1 n+c_2(n-1)+c_3(n-1)+c_4\sum_{i=2}^{n} t_i+c_5\sum_{i=2}^{n}(t_i-1)+c_6\sum_{i=2}^{n}(t_i-1)+c_7(n-1)$$

- Nel caso pessimo $t_i = i$

(elementi in ordine decrescente ➔ tutti gli elementi che precedono a[i] sono più grandi di a[i])

$$T(n)=c_1 n+c_2(n-1)+c_3(n-1)+c_4\sum_{i=2}^{n} i+c_5\sum_{i=2}^{n}(i-1)+c_6\sum_{i=2}^{n}(i-1)+c_7(n-1)$$

Analisi di InsertionSort nel caso pessimo

$$T(n)=c_1 n+c_2(n-1)+c_3(n-1)+c_4\sum_{i=2}^{n} i+c_5\sum_{i=2}^{n}(i-1)+c_6\sum_{i=2}^{n}(i-1)+c_7(n-1)$$

$$c_1+c_2+c_3+c_7)n-c_2-c_3-c_7+(c_4+c_5+c_6)\sum_{i=2}^{n} i-(c_5+c_6)(n-1)$$

$$c_1+c_2+c_3+c_7)n-c_2-c_3-c_7+(c_4+c_5+c_6)\left(\frac{n(n+1)}{2}-1\right)-(c_5+c_6)(n-1)$$

$$c_4+c_5+c_6)\frac{n^2}{2}+\left(c_1+c_2+c_3+\frac{c_4}{2}-\frac{c_5}{2}-\frac{c_6}{2}+c_7\right)n-c_2-c_3-c_7$$

$$=an^2+bn+c$$

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

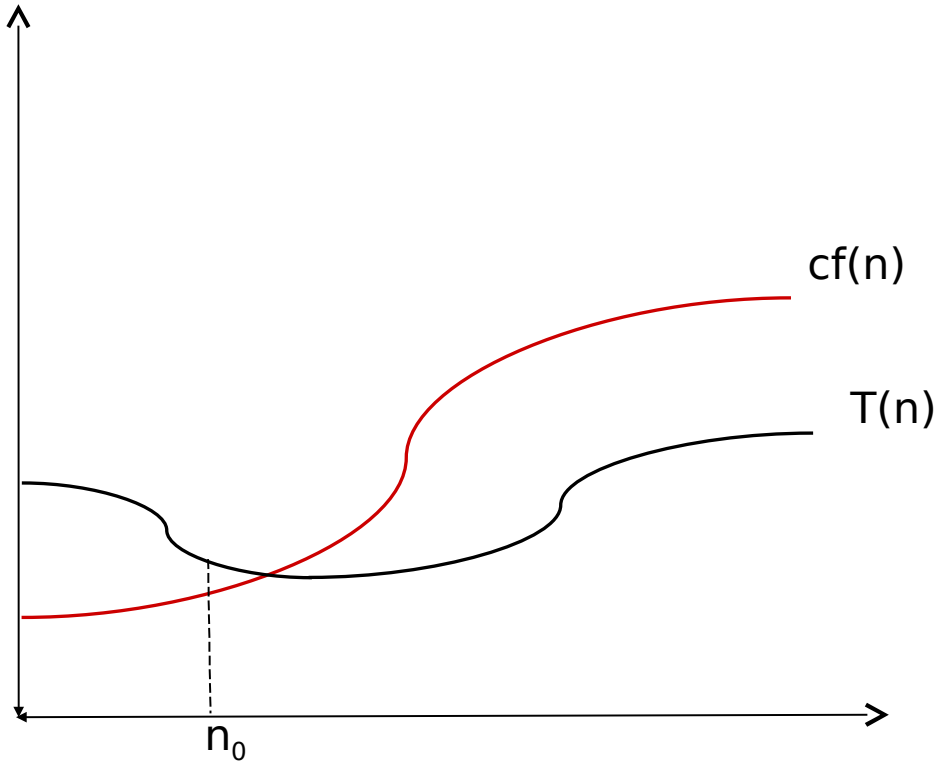
- Possiamo aumentare il livello di astrazione considerando solo l'ordine di grandezza
  - Consideriamo solo il temine "dominante"
    - Per InsertionSort: an^2
    - Giustificazione: più **grande** è n, minore è il contributo dato dagli altri termini alla stima della complessità
  - Ignoriamo del tutto 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

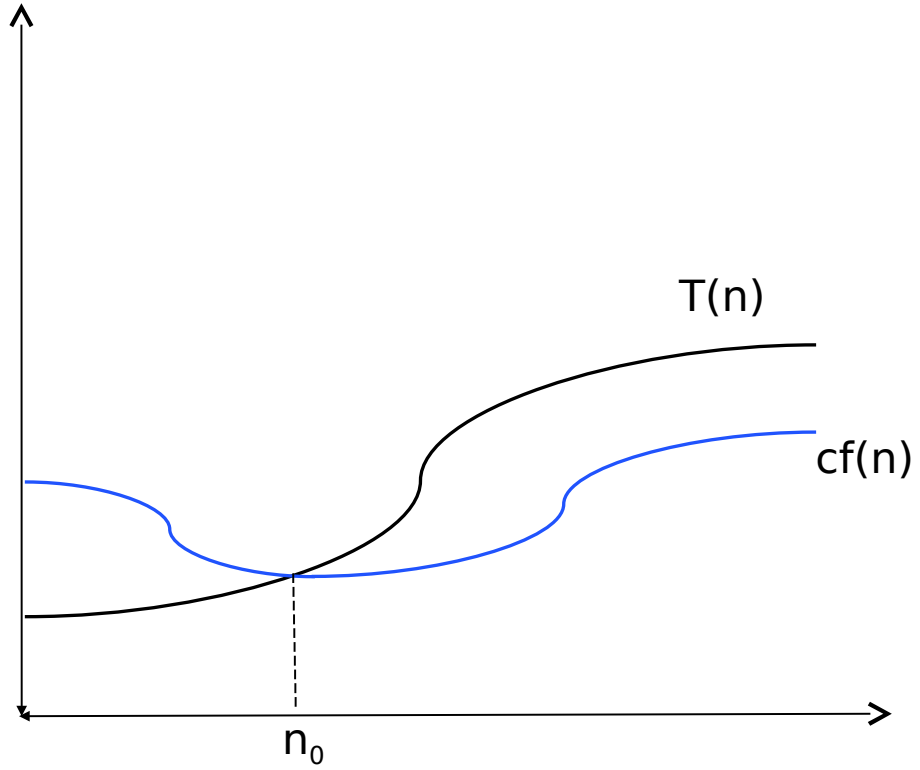
Ordine asintotico di grandezza

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 $T(n) \leq c \cdot f(n)$.



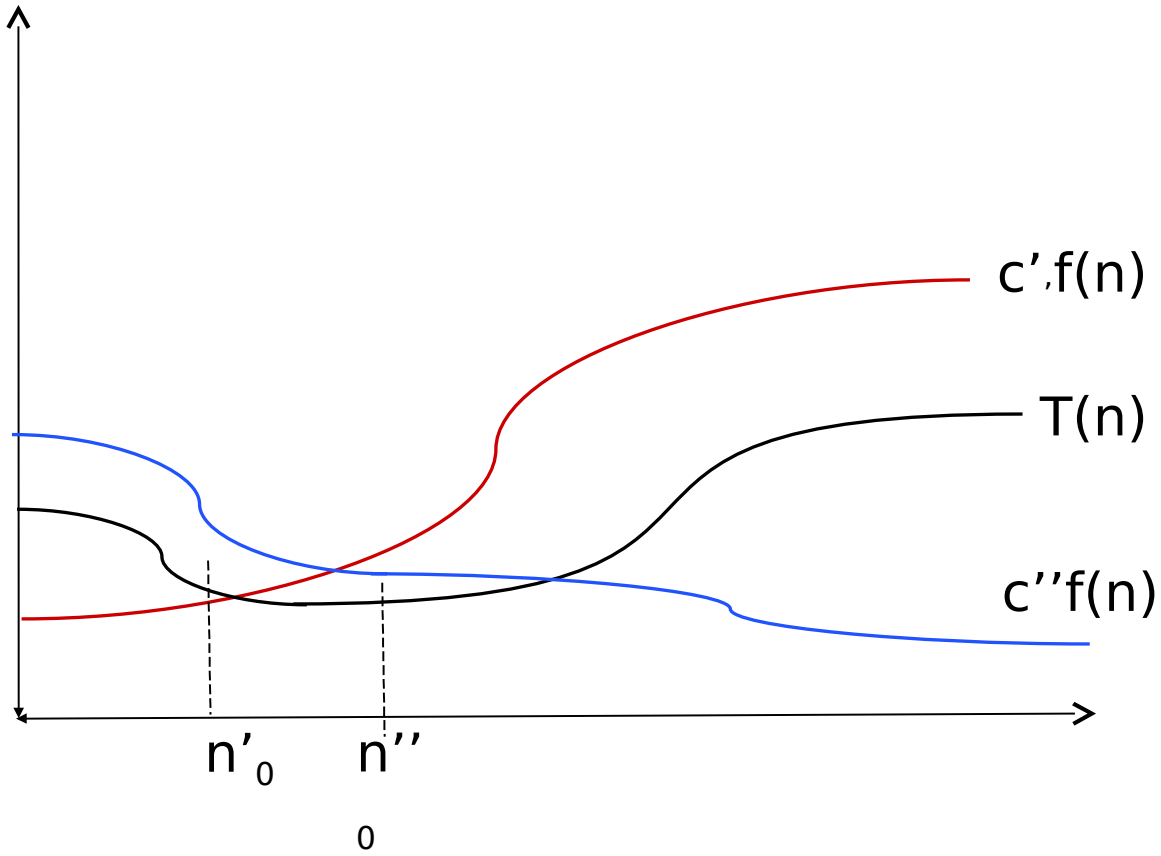
Ordine asintotico di grandezza

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 $T(n) \geq c \cdot f(n) > 0$.



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 Ω**

Affermazione corretta**.** Ogni algoritmo basato sui confronti richiede almeno Ω(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 $g = O(h)$ allora $f + g = \Theta(h)$.

Bound asintotici per alcune funzioni di uso comune

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

Dim. $O(n^d)$: Basta prendere $n_0=1$ e come costante c la somma

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

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

Bound asintotici per alcune funzioni di uso comune

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

Dim $\Omega(n^d)$:

$a_0 + a_1n + \ldots + a_dn^d \geq a_dn^d - (|a_0| + |a_1|n + \ldots + |a_{d-1}|n^{d-1})$

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 $n \geq n'_0$

Vogliamo trovare $n_0 \geq 0$ 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 $c \leq a_d - c'/n$. Siccome deve essere $c > 0$, imponiamo $a_d - c'/n > 0$ che è soddisfatta per $n > c'/a$

Prendendo quindi $n_0 = 2c'/a_d$ si ha che per $n \geq n_0$, risulta $a_d - c'/n \geq a_d - a_d/2 = a_d/2$

In conclusione abbiamo $n_0 = 2c'/a_d$ e $c = a_d/2$

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 $\Theta(n^3)$.

Bound asintotici per alcune funzioni di uso comune

- Logaritmi. log n= O(n).

  Dim. Dimostriamo per induzione che $\log_2 n \leq n$ per ogni $n \geq 1$.

  Base dell'induzione: Vero per n=1.

  Passo Induttivo: Supponiamo vero per n e dimostriamo per n+1.

  $\log_2(n+1) \leq \log_2 (2n) = \log_2 2 + \log_2 n = 1 + \log_2 n$

  Per ipotesi induttiva $\log_2 n \leq n$ e quindi $1+\log_2 n \leq n+1$.

Bound asintotici per alcune funzioni di uso comune

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

log n cresce non piu` velocemente
di ogni potenza di n

Dim. Se $x \geq 1$ si ha $n = O(n^x)$ e quindi la tesi discende dal fatto che

$\log n \leq n$ per ogni $n \geq 1$.

Consideriamo il caso $0 < x < 1$. Vogliamo trovare le costanti $c > 0$ e $n_0 \geq 0$ tali che $\log n \leq c n^x$ 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 $\log n^x \leq n^x$ da cui $x \log_2 n < n^x$

e dividendo entrambi i membri per x si ha $\log n < 1/x\, n^x$ per cui basta prendere $c = 1/x$ e $n_0 = 1$ perché la disequazione $\log n \leq c\, n^x$ sia soddisfatta.

Bound asintotici per alcune funzioni di uso comune

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

perché $\log_a n = (\log_a b)\ (\log_b n)$ e $\log_a b$ è una costante

Quindi nelle notazioni asintotiche possiamo evitare di specificare la base del logaritmo

$\log n^b = \Theta(\log n)$ per ogni costante b > 0.

perché $\log n^b = b \log n$ e b è una costante

Da quanto visto prima si ha che $\log n^b = O(n^x)$ per ogni x>0

Potenze arbitrarie di logaritmi versus potenze arbitrarie di n

Logaritmi. Per ogni e $a>0, b>0, y> 0$, $\log^a n^b = O(n^y)$. (N.B. y può essere < 1)

Qualsiasi potenza di log cresce non piu` velocemente di ogni potenza di n

Dim. Se $a=1$ discende dal fatto che $\log n^b = O(n^y)$ (gia` dimostrato)

Dimostriamolo per $a>0$ arbitario.

Abbiamo gia` dimostrato che $\log n^b = \Theta(\log n)$ e $\log n = O(n^{y/a})$ per ogni $y/a>0$

Per cui $\log n^b = O(n^{y/a})$ perche' $\Theta \rightarrow O$ e per la transitivita`

Applicando la definizione di O si ha che esistono costanti $c>0$ ed $n_0 \geq 0$ tali che $\log n^b \leq c\, n^{y/a}$ per ogni $n \geq n_0$

Elevando ad a entrambi i membri $(\log n^b)^a \leq (c\, n^{y/a})^a$ per ogni $n \geq n_0$

che equivale a scrivere $\log^a n^b \leq c^a n^y$ per ogni $n \geq n_0$

Quindi prendendo $c'=c^a$ e $n'_0 = n_0$ abbiamo $\log^a n^b \leq c' n^y$ per ogni $n \geq n'_0$

- Si ha

$$\boxed{\forall \text{ costanti } a > 0, b > 0, k > 0 \text{ vale } \log^a n^b = O(n^k)} \quad (2)$$

Ad esempio, $\log^5 n^6 = O(\sqrt[3]{n})$

Basta infatti porre $a = 5, b = 6, k = 1/3$ nella (2)

Altro esempio: $\log^2 n^{10} = O(\sqrt[6]{n})$

Basta porre $a = 2, b = 10, k = 1/6$ nella (2)

Esempi

$$n^3 + 100n + 200 = O(n^3)$$

$$20n^3 + n^5 + 100n = O(n^5)$$

$$10n^2 + n^{5/2} + 7n = O(n^{5/2})$$

$$10n+3n^7+5n^6+9n^3+34n^2+22n^5+n^{8/3}+4n^{7/2}+23n^{11/2} = O(n^7)$$

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

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.: $n^{100} = O(2^n)$

Ricapitolazione dei più comuni tempi di esecuzione

Tempo lineare: O(log n)

- Esempio: Algoritmo di ricerca binaria: cerca un numero x in un array a ordinato

```
RicercaBinaria(a,x)
primo ← 0;
ultimo ← n-1;
while (primo<ultimo)    //al massimo |log(n)|+1 volte
   centro=(primo+ultimo)/2; //il corpo al massimo|log(n)| volte
   if( x< centro)  ultimo=centro-1;
  else if( x> centro)  primo=centro+1;
  else return centro;
}
```

Tempo lineare: O(n)

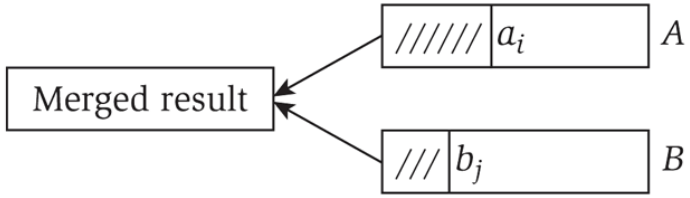
Tempo lineare. Il tempo di esecuzione è al più un fattore costante per la dimensione dell'input.

Esempio:
Computazione del massimo. Computa il massimo di n numeri a_1, ..., 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 (a_i ≤ b_j) aggiungi a_i alla fine della lista output e incrementa i

   else aggiungi b_j 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 O(n log n)

Tempo O(n log n). Viene fuori quando si esamina la complessita di algoritmi basati sulla tecnica del divide et impera (studieremo meglio questo paradigma in seguito)

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, …, 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 crescente 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 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. Esiste un algoritmo basato su divide et impera molto piu` efficiente: O(n log n)

Cubic Time: $O(n^3)$

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

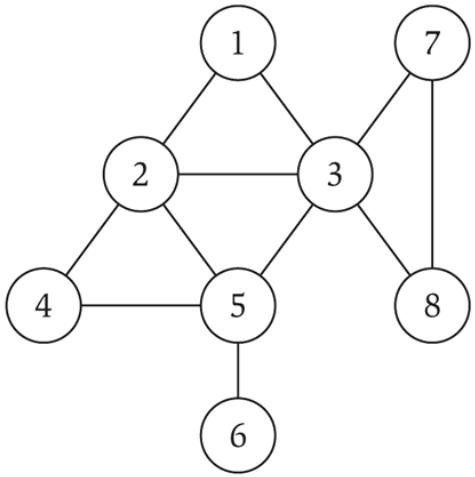
Disgiunzione di insiemi. Dati n insiemi S_1, ..., S_n ciascuno dei quali è un sottoinsieme di {1, 2, ..., n}, c'è qualche coppia di insiemi che è disgiunta?

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)

```
foreach insieme Si {
   foreach altro insieme Sj {
      foreach elemento p di Si {
         determina se p appartiene anche a Sj
      }
    If(nessun elemento di Si appartiene a Sj)
     riporta che Si  e  Sj sono disgiunti
   }
}
```

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: tempo $O(n^k)$

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.

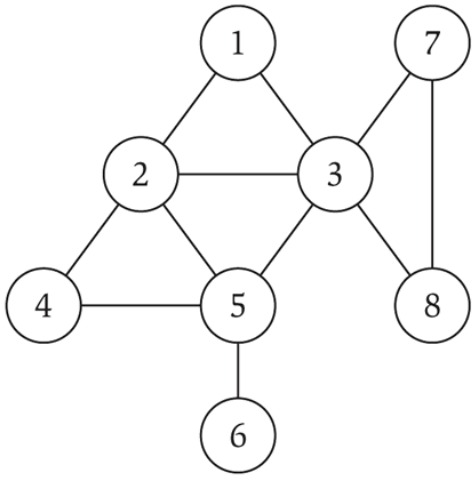
```
foreach sottoinsieme S di k nodi {
   controlla se S è un insieme indipendente
   if (S è un insieme indipendente)
      riporta che S è in insieme indiependente
   }
}
```

$$\binom{n}{k} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{k(k-1)(k-2)\ldots(2)(1)} \leq \frac{n^k}{k!}$$

- Controllare se S è un insieme indipendente = $O(k^2)$.
- Numero di sottoinsiemi di k elementi =
- $O(k^2 n^k / k!) = O(n^k)$.

Insieme indipendete

- 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 $\Theta(n^2 2^n)$. Esamina tutti i sottoinsiemi di vertici.

```
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
   }
}
```