Ricominciamo da quanto fatto

Problemi incontrati		Algoritmi
Max Subarray	Stable Matching	
n^2 possibilità	$n!$ possibilità	*Algoritmo Brute Force*
$\sim n$ operazioni	$\sim n^2$ operazioni	*Migliore Algoritmo visto*
$\sim n\log(n)$ operazioni		
$\sim n^2$ operazioni		

Stime ragionevoli del comportamento dei programmi.
Dal confronto dei running time dei programmi corrispondenti

Q. In che senso queste funzioni stimano abbastanza bene il comportamento reale?

Q. Perché ci accontentiamo di tali stime?

Forniscono l'ordine di crescita asintotica del running time

Riflettiamo su quanto fatto

Problemi incontrati		Algoritmi
Max Subarray	Stable Matching	
n^2 possibilità	$n!$ possibilità	*Algoritmo Brute Force*
$\sim n$ operazioni	$\sim n^2$ operazioni	*Migliore Algoritmo visto*

Es. di Ordine di crescita. Di quanto aumenta il running time se la taglia dell'input raddoppia?

- $\sim n^2$ dice che il running time aumenta di un fattore 4
  - $n_1 = 10$ --> $n_2 = 20$ $(n_1)^2 = 100$, $(n_2)^2 = 400 = 4x(n_1)^2$
- $\sim n$ dice che il running time aumenta di un fattore 2
  - $n_1 = 10$ --> $n_2 = 20$ $(n_2) = 2x(n_1)$
- $\sim n!$ dice che il running time aumenta di un fattore > n^n
  - $n_1 = 10$ --> $n_2 = 20$ $n_1! = 10x9x...x2x1$ $n_2! = 20x19x...x11x10!$

Un criterio di efficienza

Dicotomia tipica tra esponenziale e polinomiale.

Esiste un algoritmo naturale che prova tutte le possibili soluzioni (brute force)

- in molti casi esistono 2^n possibilità (o di più) per un input di taglia n.
- un tale algoritmo è, in pratica, inaccettabile
- l'ordine di crescita del running time è 2^n
  - *se l'input raddoppia il running time cresce del quadrato*

Proprietà di crescita desiderabile. Se la taglia dell'input raddoppia, il running time dovrebbe peggiorare di un fattore costante C (indipendente dalla taglia dell'input)

Un criterio di efficienza

Proprietà di crescita desiderabile. Se la taglia dell'input raddoppia, il running time dovrebbe peggiorare di un fattore costante C (indipendente dalla taglia dell'input)

Def. Un algoritmo è polinomiale (tempo polinomiale) se vale la proprietà di crescita di cui sopra.

Esistono costanti c > 0 e d > 0 tali che su ogni input di taglia n, il running time è quello di al più $c\,n^d$ passi computazionali elementari

← scegliamo $c = 2^d$

Running time polinomiale

Diremo che un algoritmo è efficiente se il running time è polinomiale.

Esistono costanti c > 0 e d > 0 tali che
su ogni input di taglia n, il running time è quello di
al più c n^d passi computazionali elementari

Perché proprio questa definizione?. Tale scelta funziona in pratica!

- In pratica, gli algoritmi polinomiali sviluppati fino ad ora hanno costanti ed esponenti piccoli.
- Superare la barriera esponenziale dell'algoritmo esaustivo (*brute force*) rivela elementi strutturali del problema (fa capire di più).

Q. Quale running time considereresti *più efficiente*

$20\,n^{100}$ oppure $n^{1+0.02\ln n}$?

Notazioni Asintotiche: O (o-grande), Ω (omega), e Θ (teta)

Prossimo passo. valutare e confrontare l'ordine di crescita asintotico di funzioni elementari.

Uso finale. stima della velocità di crescita di funzioni che esprimono le risorse consumate da un algoritmo rispetto alla taglia dell'input

Perché analisi asintotica.

- Perché di solito usiamo gli algoritmi per risolvere istanze di taglia grande
- Perché le istanze piccole potrebbero essere una sorta di eccezione rispetto al comportamento tipico di un algoritmo

Vogliamo poter ragionare come segue:
Algo *A* cresce come *f(n)*
Algo *B* cresce come *g(n)*
f(n) cresce meno velocemente di *g(n)*
quindi *A* è più efficiente di *B*

Perché un tale approccio dovrebbe funzionare?

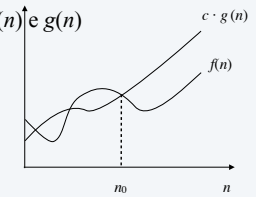
Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	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

Notazione O-grande (Big-Oh notation)

Usata per limitare superiormente il tasso di crescita

Def. Date due funzioni (asintoticamente non negative) $f(n)$ e $g(n)$
$f(n)$ è $O(g(n))$ [scritto anche $f(n) = O(g(n))$]
se esistono constanti $c > 0$ e $n_0 \geq 0$
tali che $f(n) \leq c \cdot g(n)$ per ogni $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ è $O(n^2)$. ← scegliendo $c = 50$, $n_0 = 1$
- $f(n)$ è anche $O(n^3)$.
- $f(n)$ non è $O(n)$ né $O(n \log n)$.

- $\log n$ è $O(n)$
- $(\log n^k)^m$ è $O(n^a)$ per ogni k, m e $a > 0$ *[es. $(\log n^{10})^7$ è $O(n^{0.0001})$]*
- n^k è $O(a^n)$ per ogni k e a *[es. n^{1000} è $O(0.0001^n)$]*
- a^n è $O(n!)$ per ogni a

Limiti asintotici di alcune funzioni elementari

Polinomi. Sia $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ con $a_d > 0$. Allora, $f(n)$ è $O(n^d)$.

Dim.

$$\begin{aligned} a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 &\leq |a_d| n^d + |a_{d-1}| n^{d-1} + \cdots + |a_1| n + |a_0| \\ &\leq (|a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0|)\, n^d \\ &= c n^d \end{aligned}$$

quindi $n_0 = 1$ e $c = |a_d| + \ldots + |a_0|$ soddisfano la definizione

Limiti asintotici di alcune funzioni elementari

Logaritmi e polinomi. Abbiamo ***log n = O(n)***--- assumiamo che il logaritmo sia in base 2

Dim. Possiamo dimostrare per induzione che per ogni $n > 0$, vale $\log n \leq cn$

Per $n=1$ abbiamo $\log n = 0 < 1 = n$

Assumendo per ipotesi induttiva che valga $\log (n-1) \leq c\,(n-1)$, otteniamo

$\log n = \log (n-1+1) \leq 1 + \log (n-1) \leq 1 + c(n-1) \leq c\, n$

Logaritmi e polinomi. Per ogni $k, m, \alpha > 0$, vale **$(\log n^m)^k$ is $O(n^\alpha)$**.

Dim

$$\log_2 n^m = m \log_2 n = \frac{mk}{\alpha} \times \frac{\alpha}{k} \log_2 n = \frac{mk}{\alpha} \log_2 n^{\frac{\alpha}{k}} \leq \frac{mk}{\alpha} \times d n^{\frac{\alpha}{k}}$$

da cui otteniamo (ponendo $c = (mkd/\alpha)^k$)

$$\log_2^k n^m \leq \left(\frac{mk}{\alpha} \times d \right)^k n^\alpha = c n^\alpha$$

Limiti asintotici di alcune funzioni elementari

Logaritmi: la base non conta, cambia solo la costante.

$$\log_a n = \frac{\log_b n}{\log_b a}$$

quindi, in termini di crescita asintotica, quanto dimostriamo per $\log_b(n)$ vale anche per $\log_a(n)$

Esponenziali e polinomi. Per ogni $a > 1$ e ogni $k > 0$, **n^k è $O(a^n)$**.

Dim. Sia n_0 il minimo n tale che $\frac{n}{\log_a n} \geq k$

quindi per ogni $n > n_0$ otteniamo che

$$a^n = \left(a^{\frac{n}{\log_a n}}\right)^{\log_a n} = \left(a^{\log_a n}\right)^{\frac{n}{\log_a n}} = n^{\frac{n}{\log_a n}} \geq n^k.$$

Abusi di notazione

Uso del segno uguale. $O(g(n))$ è un insieme di funzioni, ma spesso si scrive

$f(n) = O(g(n))$ invece di $f(n) \in O(g(n))$.

Errori da evitare. Sia $f(n) = 5n^3$ e $g(n) = 3n^2$.

- Se scriviamo $f(n) = O(n^3) = g(n)$.
- Allora (!?!), $f(n) = g(n)$.

Dominio. Il dominio delle funzioni è quello dei numeri naturali $\{0, 1, 2, \ldots\}$.

- a volte restringeremo ad un sottinsieme dei naturali
- altre volte estenderemo ai reali

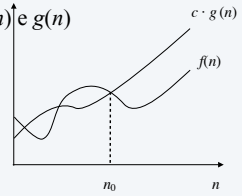
Funzioni non-negative. Quando si usa la notazione O-grande, assumiamo che le funzioni siano (asintoticamente) nonnegative.

IMPORTANTE. OK ad abusi di notazione; attenzione agli eccessi erronei!

Ieri abbiamo introdotto la notazione O-grande (Big-Oh notation)

Usata per limitare superiormente il tasso di crescita

Def. Date due funzioni (asintoticamente non negative) $f(n)$ e $g(n)$
$f(n)$ è $O(g(n))$ [scritto anche $f(n) = O(g(n))$]
se esistono costanti $c > 0$ e $n_0 \geq 0$
tali che $f(n) \leq c \cdot g(n)$ per ogni $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$.

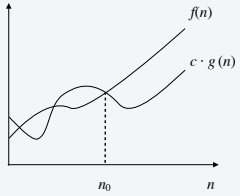
- $f(n)$ è $O(n^2)$. ← scegliendo $c = 50$, $n_0 = 1$
- $f(n)$ è anche $O(n^3)$.
- $f(n)$ non è $O(n)$ né $O(n \log n)$.
- **c è O(1) (ogni costante è "equivalente" ad 1)**
- $\log n$ è $O(n)$
- $(\log n^k)^m$ è $O(n^a)$ per ogni k, m e $a > 0$ *[es. $(\log n^{10})^7$ è $O(n^{0.0001})$]*
- n^k è $O(a^n)$ per ogni k e a *[es. n^{1000} è $O(0.0001^n)$]*
- a^n è $O(n!)$ per ogni a

Notazione Ω (Big-Omega notation)

Per Limiti inferiori. $f(n)$ è $\Omega(g(n))$ se esistono costanti $c > 0$ e $n_0 \geq 0$
tali che $f(n) \geq c \cdot g(n)$ per ogni $n \geq n_0$.

Ea. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ è sia $\Omega(n^2)$ che $\Omega(n)$. ← per c = 32, n_0 = 1
- $f(n)$ non è $\Omega(n^3)$ né $\Omega(n^3 \log n)$.



Uso. La notazione *O* servirà per esprimere limiti superiori alle risorse usate da un algoritmo. La notazione Ω servirà per esprimere limiti inferiori alle risorse usate da un algoritmo.

Affermazioni da evitare. Ogni algoritmo di ordinamento basato su confronti richiede almeno $O(n \log n)$ confronti nel caso peggiore.

Ω() e O() sono duali

Dualità.

- $f(n)$ è $\Omega(g(n))$ se e solo se $g(n)$ è $O(f(n))$

basta notare che $f(n) = \Omega(g(n))$ se e solo se esistono costanti $c, n_0 > 0$
tali che $f(n) \geq c\, g(n)$

quindi se e solo se, per $c' = 1/c$, vale $g(n) \leq c'\, f(n)$ cioè $g(n) = O(f(n))$

- $c = O(1)$
- $\log\log n = O(\log n)$
- $\log n = O(n)$
- $(\log n^k)^m = O(n^a)$ per ogni k, m e $a > 0$
- $n^k = O(a^n)$ per ogni k e a
- $a^n = O(n!)$ per ogni a

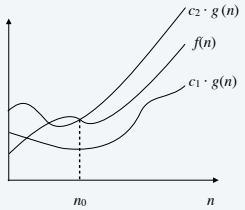
- $c = \Omega(1)$
- $\log n = \Omega(\log n)$
- $n = \Omega(\log n)$
- $n^a = \Omega(\log n^k)^m$ per ogni k, m e $a > 0$
- $a^n = \Omega(n^k)$ per ogni k e a
- $n! = \Omega(a^n)$ per ogni a

Notazione Θ (Theta-notation)

Limiti inferiori e superiori. $f(n)$ è $\Theta(g(n))$ se esistono costanti $c_1 > 0$, $c_2 > 0$, $n_0 \geq 0$
tali che $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ **for all** $n \geq n_0$.

Es. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is $\Theta(n^2)$. ← con $c_1 = 32$, $c_2 = 50$, $n_0 = 1$
- $f(n)$ non è $\Theta(n)$ né $\Theta(n^3)$.



Uso. L'algoritmo *maxsubarray3* esegue $\Theta(n \log n)$ operazioni su un array di n elementi.

Nota. $f(n)$ è $\Theta(g(n))$ se e solo se valgono entrambe

- **$f(n)$ è $O(g(n))$**
- **$f(n)$ è $\Omega(g(n))$**

Regole Utili (le mostriamo per O, ma valgono anche per Ω e Θ)

Le costanti non contano.

- Se $f(n) = O(g(n))$ allora $a\,f(n) = O(g(n))$, per ogni costante $a > 0$
- *Es.* log $n = O(n)$ *allora anche* $10 \log n = O(n)$

Somme.

- Se $e(n) = O(g(n))$, $f(n) = O(h(n))$, allora $e(n) + f(n) = O(g(n) + h(n))$
- *Es.* log $n = O(\sqrt{n})$, $3n^2 = O(n^2)$ *allora anche* $\log n + 3n^2 = O(\sqrt{n} + n^2)$

Prodotti.

- Se $e(n) = O(g(n))$, $f(n) = O(h(n))$, allora $e(n)\,f(n) = O(g(n)\,h(n))$
- *Es.* log $n = O(\sqrt{n})$, $3n^2 = O(n^2)$ *allora anche* $3n^2 \log n = O(n^2\sqrt{n})$

Transitività.

- Se $f(n) = O(g(n))$, $g(n) = O(h(n))$, allora $f(n) = O(h(n))$
- *Es.* log $n = O(\sqrt{n})$, $\sqrt{n} = O(n/\log n)$ *allora anche* $\log n = O(n/\log n)$

Regole Utili (le mostriamo per O, ma valgono anche per Ω e Θ)

Somma e transitività per O

- Se $f(n) = O(g(n))$, $d(n) = O(e(n))$, e $g(n) = O(e(n))$ allora $f(n) + d(n) = O(e(n))$
- *Es.* log $n = O(\sqrt{n})$, $3n^2 = O(n^2)$ *allora anche* $\log n + 3n^2 = O(n^2)$

Somma e transitività per Ω

- Se $f(n) = \Omega(g(n))$, $d(n) = \Omega(e(n))$, e $g(n) = O(e(n))$ allora
  - $f(n)+d(n) = \Omega(g(n))$ e anche $f(n)+d(n) = \Omega(e(n))$
- *Es.* $\sqrt{n} = \Omega(\log n)$, $n \log n = \Omega(n)$ allora
  - $\sqrt{n} + n \log n = \Omega(n)$
  - $\sqrt{n} + n \log n = \Omega(\log n)$

Altri metodi utili per verificare ordini crescita

Proposizione. Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$, allora $f(n)$ è $\Theta(g(n))$.

DIM. Per definizione di limite, esiste n_0 tale che per ogni $n \geq n_0$

$$\frac{1}{2}c < \frac{f(n)}{g(n)} < 2c$$

- Quindi, $f(n) \leq 2\,c\,g(n)$ per ogni $n \geq n_0$, il ché implica $f(n)$ è $O(g(n))$.
- Analogamente, $f(n) \geq \frac{1}{2}\,c\,g(n)$ per ogni $n \geq n_0$, quindi $f(n)$ is $\Omega(g(n))$.

Proposizione. Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, allora $f(n)$ è $O(g(n))$.

Notazione O-grande per funzioni a più variabili

Limiti superiori. $T(m,n)$ è $O(f(m,n))$ se esistono costanti $c > 0$, $m_0 \geq 0$, e $n_0 \geq 0$ tali che $T(m,n) \leq c \cdot f(m,n)$ per ogni $n \geq n_0$ e $m \geq m_0$.

Es. $T(m,n) = 32mn^2 + 17mn + 32n^3$.

- $T(m,n)$ è sia $O(mn^2 + n^3)$ che $O(mn^3)$.
- $T(m,n)$ non è $O(n^3)$ né $O(mn^2)$.

Esempi di uso. Visita per ampiezza di un grafo (breadth-first search) cammino minimo tra due vertici in un grafo diretto.

Analisi di complessità di un algoritmo

Ci concentriamo sulla risorsa TEMPO
- tempo di esecuzione in funzione della taglia dell'input

Tempo di esecuzione
- numero di operazione elementari eseguite sull'istanza
- quale istanza?

Taglia dell'input
- come la misuriamo?

Funzione della taglia dell'input
- useremo le notazioni asintotiche

Tempo di esecuzione: operazioni elementari

Numero di operazioni elementari eseguite
- operazioni aritmetiche, confronto tra elementi, seguire un puntatore, etc.
- assumiamo che ogni operazione elementare richieda tempo $\Theta(1)$

Calcolo della complessità (di tempo)
- operazioni di assegnamento di variabili: costo $\Theta(1)$
- istruzioni condizionali (**if ... then ... else**): costo pari al tempo del test (in genere $\Theta(1)$) più il costo dell'alternativa più costosa
- istruzioni di iterazione (loop **while**, **repeat**, **for**) : costo pari alla somma su tutte le iterazioni del costo delle istruzioni eseguite nell'iterazione

Funzioni ricorsive
- analisi delle ricorrenze (lo vedremo quando servirà)
  - esempio: maxsubarray2 (albero di ricorsione)

Tempo di esecuzione: Quale istanza usiamo

Il running time di un algoritmo può dipendere dalla forma dell'input
- es. ordinare una lista già ordinata
- cercare un elemento scorrendo una lista
  - l'elemento è il primo
  - l'elemento non è presente

Worst case. Limiti sul running time che valgono per ogni input di taglia n.
- In genere coincide con l'efficienza verificata in pratica.
- Punto di vista pessimistico! perchè dovremmo ignorare il caso pessimo?

Eccezioni. Esistono alcuni algoritmi esponenziali usati in pratica perché le istanze che determinano il caso pessimo (esponenzialità) sembrano essere rare.

simplex algorithm

Tempo di esecuzione: analisi alternativa - caso medio

Worst case. Limiti al running time per ogni input di taglia n.
Es. Trovare il massimo di n elementi richiede al massimo n confronti.

Caso Medio. Expected running time per un input random di taglia n.
Problemi. quale distribuzione di probabilità va considerata?

Taglia dell'input

Numero di elementi che lo costituiscono

- l'input è un array o una lista: il numero di elementi
- l'input è un grafo: il numero di vertici e il numero di archi
- moltiplicazione di matrici: le dimensioni delle matrici

Il numero di bit che servono per rappresentare l'input

- moltiplicazione tra due numeri

Relazione tra le due misure

- Assumendo che
  - ogni elemento dell'input entri in una parola di memoria
  - ogni istruzione macchina richieda tempo costante
- in generale esiste una relazione tra le due misure che è data dalla taglia della parola di memoria che si utilizza

Riassumendo

L'algoritmo A ha complessità (di tempo) $T(n) = O(n^2)$ significa:

- A non richiede al più tempo cn^2 per dare l'output,
  - per un opportuna costante c
  - per ogni input di taglia n sufficientemente grande

L'algoritmo A ha complessità (di tempo) $T(n) = \Omega(n^2)$ significa:

- A richiede tempo almeno cn^2 prima di fornire l'output
  - per un opportuna costante c e
  - per almeno un input di taglia n sufficientemente grande

Sottintendiamo il caso peggiore in entrambe le affermazioni

- Il limite superiore si estende a tutti i casi
- il limite inferiore garantisce tutti i casi
- non significa che ogni caso necessita almeno del tempo dato come limite inferiore

Esempi ed esercizi

Utilizzando la notazione Θ si stimi la complessità dei seguenti stralci di codice:

1.
```
for i = 1 to 2n
  if x è pari then
     x = x-1
  else
     x = x+2
```

2.
```
for i = 1 to 3n
   for j = 9 to n
      x = x + 1
```

3.
```
for i = 1 to n
   for j = 1 to i
      for k = j to i
        x = x + 1
```


Esempi ed esercizi

Utilizzando la notazione Θ si stimi la complessità dei seguenti stralci di codice:

4.
```
i = n
while i ≥ 1 do
   x = x+1, i = i/2
```

5.
```
j = n
while j ≥ 1 do
   for i = 1 to j
      x = x + i - 1
   j = j/2
```

Esempi ed esercizi

Utilizzando la notazione Θ si stimi la complessità dei seguenti stralci di codice:

6.

```
i = n
while i ≥ 1 do
   for j = 1 to n
      x = x + 1
   i = i/2
```

7.

```
i = 2
while i < n do
   i = i * i
   x = x + 1
```


Alcune utili relazioni matematiche

Sommatorie potenze successive (progressioni)

- per $\alpha \neq 1$ vogliamo calcolare $\sum_{j=0}^{k} \alpha^j$

$$
\begin{aligned}
(\alpha-1)\sum_{j=0}^{k}\alpha^j &= \sum_{j=0}^{k}\alpha^{j+1} - \sum_{j=0}^{k}\alpha^j \\
&= \alpha^1+\alpha^2+\cdots+\alpha^n+\alpha^{n+1}-\alpha^0-\alpha^1-\alpha^n \\
&= \left(\alpha^1+\alpha^2+\cdots+\alpha^n\right)+\alpha^{n+1}-\alpha^0-\left(\alpha^1+\alpha^2+\cdots+\alpha^n\right) \\
&= \alpha^{n+1}-\alpha^0 = \alpha^{n+1}-1.
\end{aligned}
$$

- quindi $\sum_{j=0}^{k}\alpha^j = \frac{\alpha^{n+1}-1}{\alpha-1}$

- Per α < 1, otteniamo quindi $\sum_{j=0}^{k}\alpha^j = \frac{1-\alpha^{n+1}}{1-\alpha} = \Theta(1)$
  - il numeratore è < 1
  - il denominatore è una costante

Esempi ed esercizi

Valutare la complessità del seguente algoritmo

```
Algoritmo (A[1,...n]
for i = 1 to n
  B[i] = A[i]
for i = 1 to n
  j = n
  while j > i do
       B[i] = B[i] + A{i]
       j = j - 1
for i = 1 to n
  t = t + B[i]
```


Esempi ed esercizi

Valutare la complessità del seguente algoritmo

```
Moltiplica (y, z)
  x = 0
  while z > 0 do
    if z è dispari then x = x+y
      y = 2*y
      z = z/2 // troncato all'intero
  return x
```

Altri esempi - O(n³)

Insiemi disgiunti. Dati n insiemi $S_1, \ldots, S_n$ ognuno un sottinsieme di $1, 2, \ldots, n$, esiste una coppia di insiemi disgiunti?

Soluzione. Per ogni coppia di insiemi, verifica se sono disgiunti.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```


Tempo Esponenziale

Insieme Indipendente. Dato un grafo, qual è la massima cardinalità di un insieme indipendente?

Soluzione. Enumera tutti i sottinsiemi e verifica se sono indipendenti.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

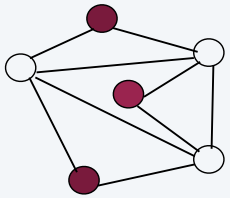
Tempo $O(n^2 2^n)$: Il numero di sottinsiemi di un insieme di n elementi è 2^n

Altri esempi

Insieme Independente di taglia k. Dato un grafo, esistono k nodi tali che nessuna coppia tra essi è unita da un arco?

k is a constant

Soluzione. Enumera tutti i sottinsiemi di k nodi.



```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Verificare se S, ($|S| = k$) è indipendente richiede tempo $O(k^2)$. (***perché?***)
- numero sottinsiemi di taglia k = $\binom{n}{k} = \frac{n(n-1)(n-2)\times\cdots\times(n-k+1)}{k(k-1)(k-2)\times\cdots\times 1} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$.

polinomiale per k=17,
ma sicuramente poco pratico