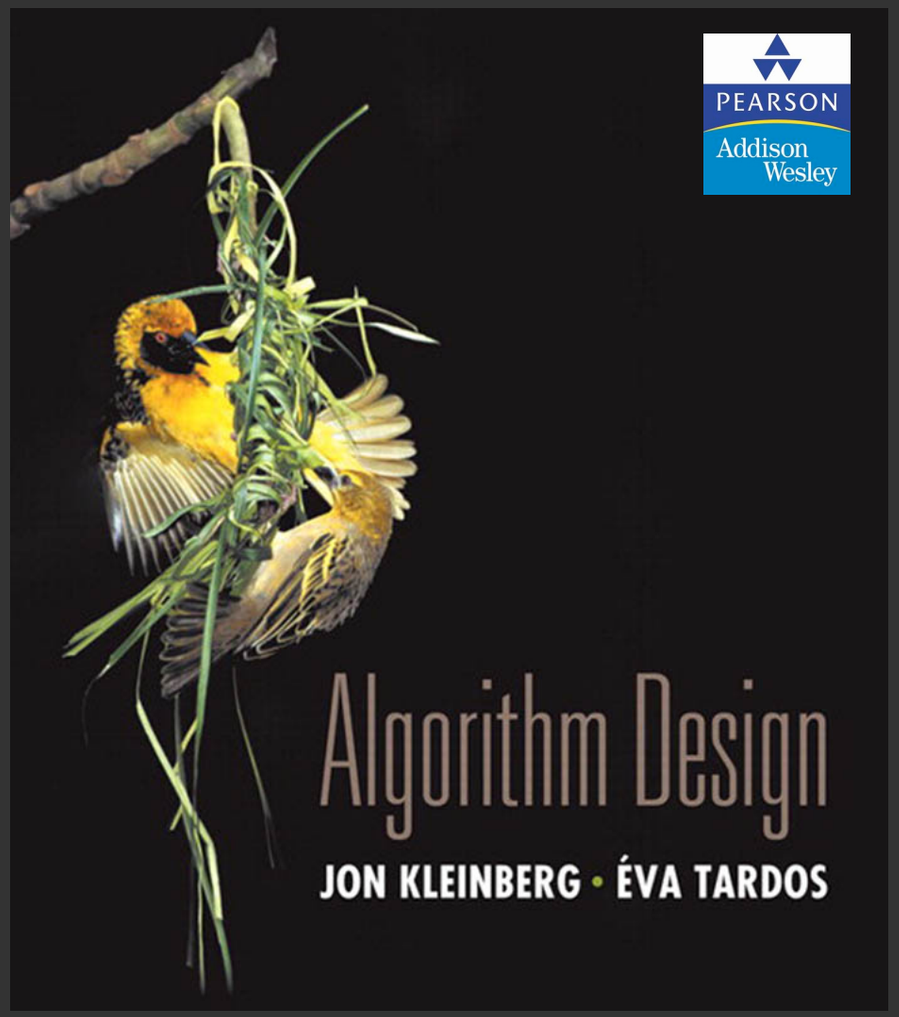


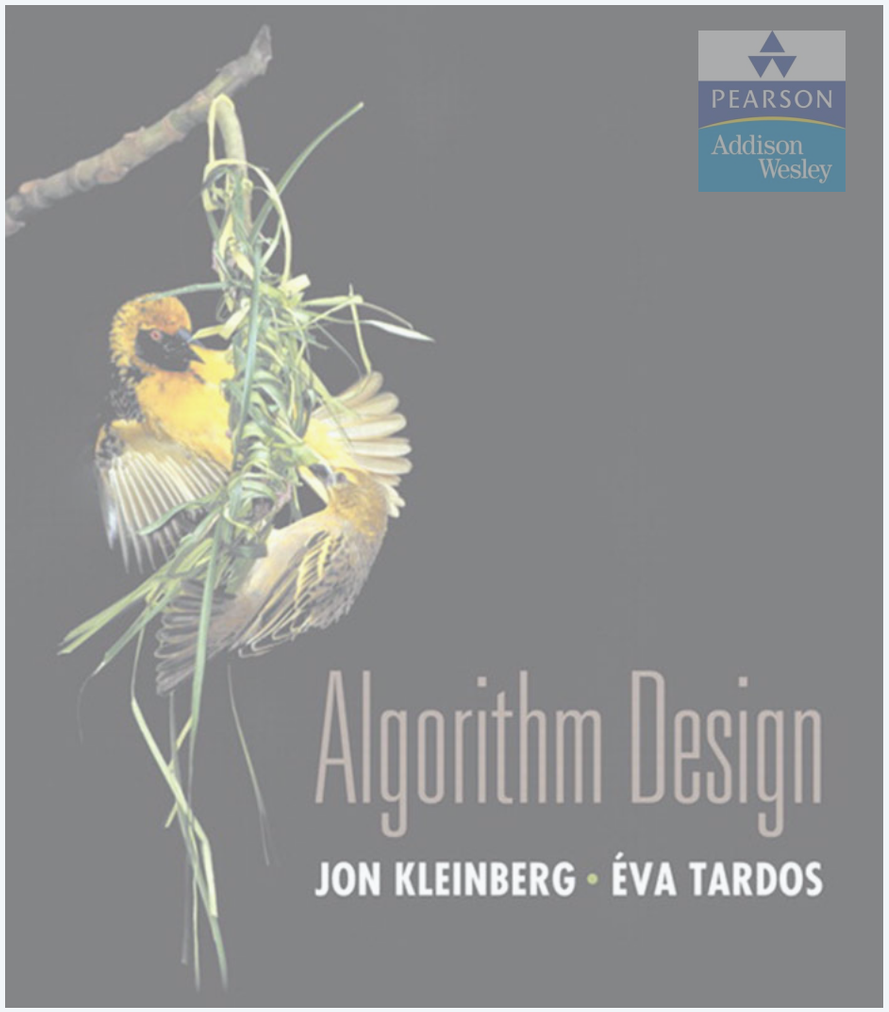
2. Analisi degli Algoritmi

- *trattabilità computazionale*
- *ordine asintotico di crescita*
- *implementare Gale–Shapley*
- *tempi di esecuzione più comuni*

Traduzione e adattamento di Vincenzo Bonifaci
Original lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 29/02/24 11:12



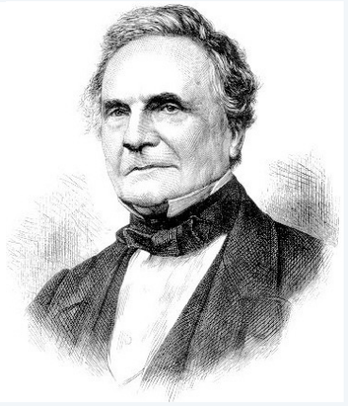
SECTION 2.1

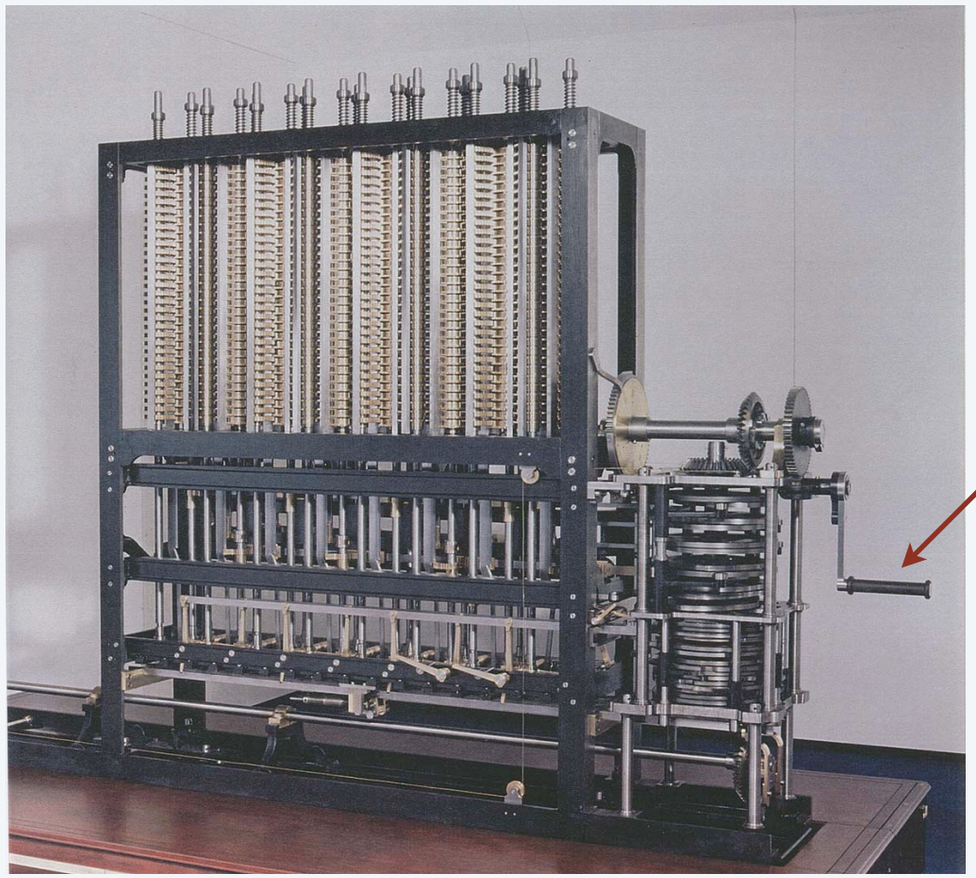
2. ANALISI DEGLI ALGORITMI

- *trattabilità computazionale*
- *ordine asintotico di crescita*
- *implementare Gale-Shapley*
- *tempi di esecuzione più comuni*

Un pensiero sorprendentemente moderno

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*



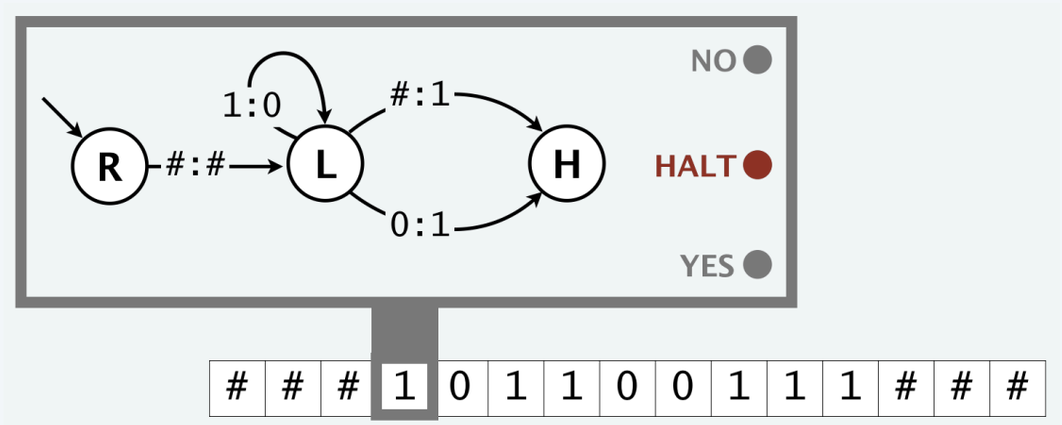


quante volte occorre
girare la manovella?

Macchina Differenziale di C. Babbage

Modelli di calcolo: macchine di Turing

Macchina di Turing deterministica. Un modello semplice ed idealizzato.



Tempo di esecuzione. Numero di passi.

Memoria. Numero di celle di nastro utilizzate.

Caveat. Non fornisce accesso casuale in memoria.

- MT ad 1 nastro richiede $\geq n^2$ passi per riconoscere palindromi a n-bit.
- Facile riconoscere tali palindromi in $\leq cn$ passi su un computer reale.

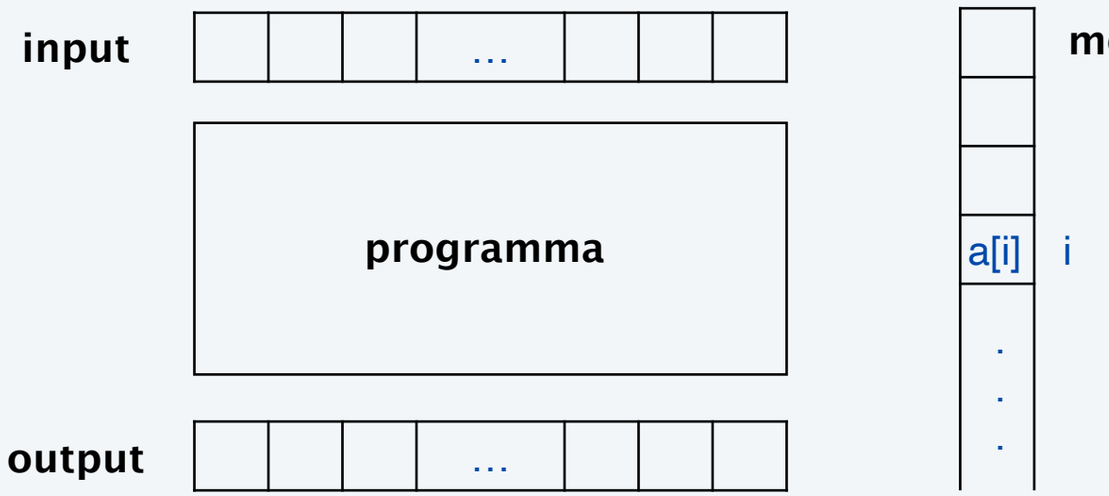
Modelli di calcolo: word RAM

assume $w \geq \log_2 n$

Word RAM.

- Locazioni di memoria e celle di I/O memorizzano interi a w-bit.
- Operazioni primitive: operazioni logico/aritmetiche, lettura/scrittura in memoria, indicizzazione di array, risoluzione di un puntatore, …

operazioni stile C in tempo costante ($w = 64$)



Tempo di esecuzione. Numero di operazioni primitive.

Memoria. Numero di celle di memoria utilizzate.

Caveat. Talvolta, si richiede un modello più preciso.

Forza bruta

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

- Tipicamente richiede 2^n passi (o peggio) per input di taglia n.
- Inaccettabile in pratica.

Es. Problema degli abbinamenti stabili: generare tutti gli $n!$ abbinamenti perfetti.

Tempo di esecuzione polinomiale

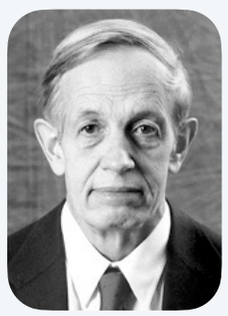
Una desiderabile proprietà di scala. Quando la taglia dell'input raddoppia, l'algoritmo dovrebbe rallentare al più di un qualche fattore moltiplicativo C.

Def. Un algoritmo è tempo-polinomiale [poly-time] se vale la suddetta proprietà di scala.

Esistono costanti a > 0 e b > 0 tali che,
per ogni input di taglia n, l'algoritmo effettua
≤ a · n^b passi computazionali primitivi.

← corrisponde a $C = 2^b$


von Neumann (1953)


Nash (1955)


Gödel (1956)


Cobham (1964)


Edmonds (1965)


Rabin (1966)

Tempo di esecuzione polinomiale

Diciamo che un algoritmo è efficiente se ha un tempo di esecuzione polinomiale.

Teoria. La definizione è (relativamente) insensibile al modello di calcolo.

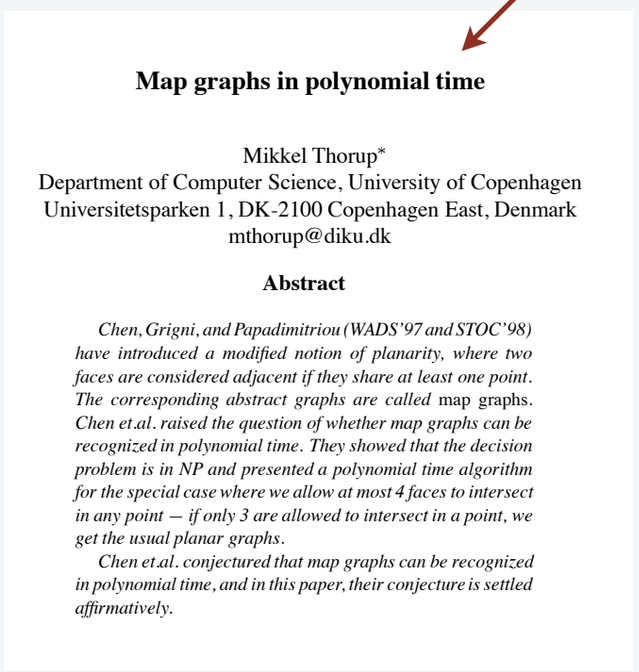
Pratica. La definizione funziona bene in pratica!

- Algoritmi tempo-polinomiali sviluppati in pratica hanno tipicamente sia costanti moltiplicative piccole che esponenti piccoli.
- Rompere la barriera esponenziale della forza bruta tipicamente rivela una struttura cruciale del problema in esame.

Eccezioni. Alcuni algoritmi tempo-polinomiali hanno costanti galattiche e/o esponenti enormi.

D. Cosa preferireste: $20\,n^{120}$ or $n^{1 + 0.02 \ln n}$?

n^{120}



Map graphs in polynomial time

Mikkel Thorup*
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
mthorup@diku.dk

Abstract

Chen, Grigni, and Papadimitriou (WADS'97 and STOC'98) have introduced a modified notion of planarity, where two faces are considered adjacent if they share at least one point. The corresponding abstract graphs are called map graphs. *Chen et.al. raised the question of whether map graphs can be recognized in polynomial time. They showed that the decision problem is in NP and presented a polynomial time algorithm for the special case where we allow at most 4 faces to intersect in any point — if only 3 are allowed to intersect in a point, we get the usual planar graphs.*

Chen et.al. conjectured that map graphs can be recognized in polynomial time, and in this paper, their conjecture is settled affirmatively.

Perché è importante

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.

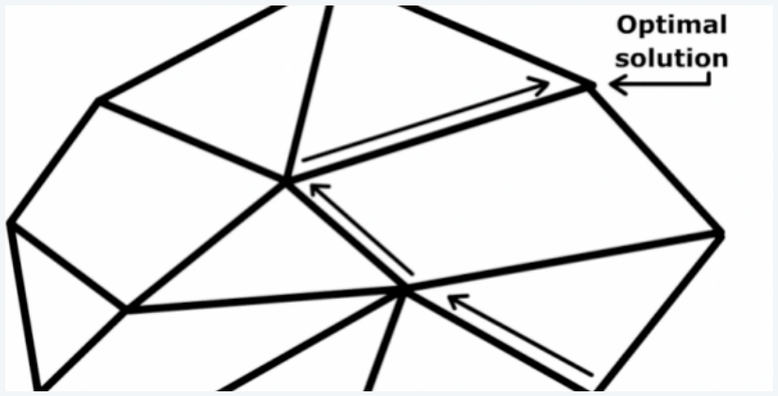
	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	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

Analisi del caso peggiore

Caso peggiore. Garanzia sul tempo di esecuzione per qualunque input di taglia n.

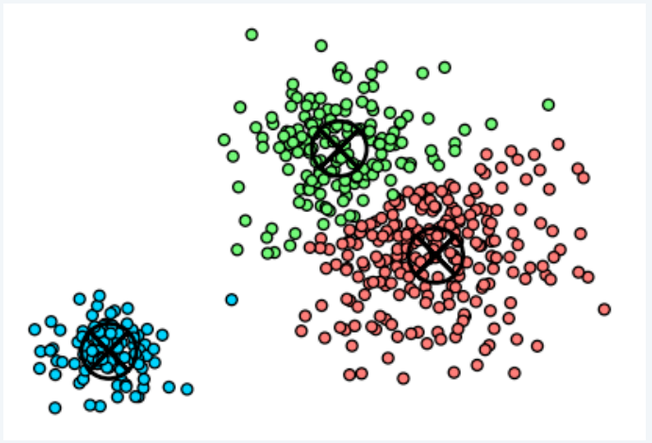
- Generalmente cattura il concetto di efficienza in pratica.
- Approccio draconiano, ma è difficile trovare alternative efficaci.

Eccezioni. Alcuni algoritmi tempo-esponenziali sono molto usati in pratica perché gli input corrispondenti al caso peggiore non si realizzano.



algoritmo del simplesso

Linux grep



algoritmo k-means

Altri tipi di analisi

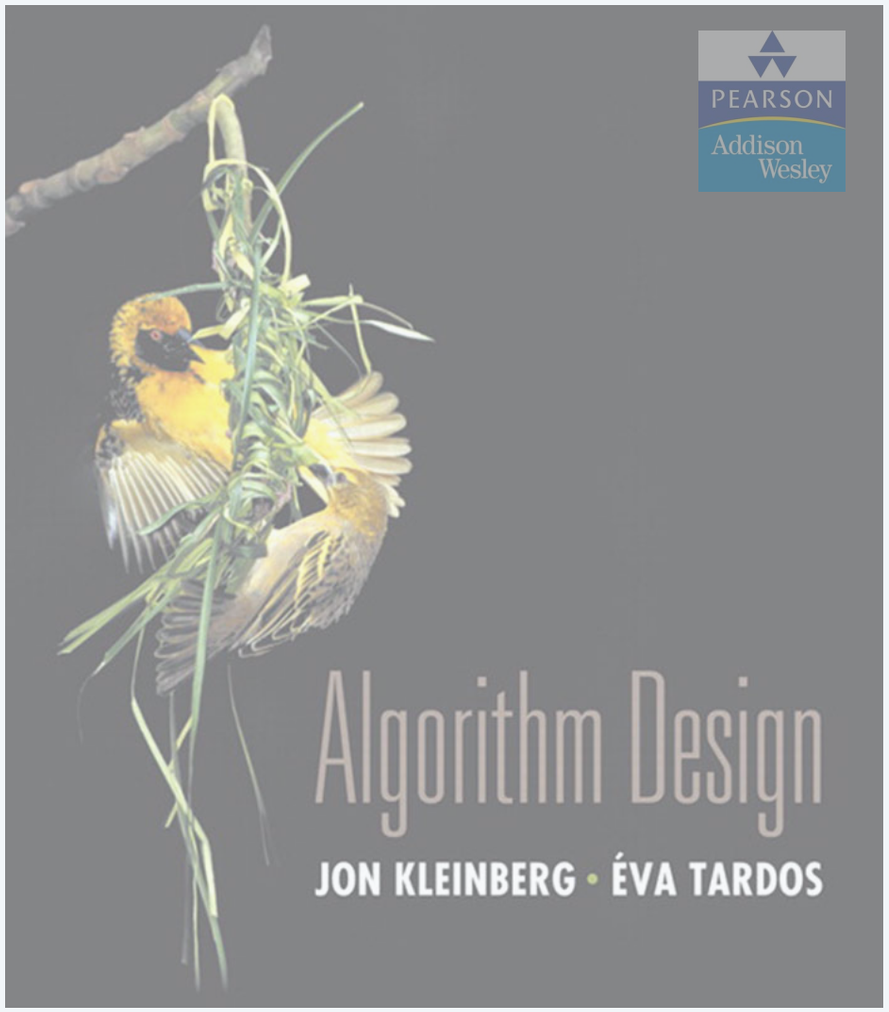
Probabilistica. Tempo di esecuzione atteso di un algoritmo randomizzato.

Es. Il numero atteso di confronti nel quicksort di n elementi è $\sim 2n \ln n$.

Ammortizzata. Il tempo di esecuzione peggiore per qualunque sequenza di n operazioni.

Es. Partendo da una pila vuota, una sequenza di n operazioni push e pop richiede $O(n)$ passi computazionali primitivi utilizzando un array dinamico.

Inoltre. Analisi del caso medio, *smoothed analysis*, analisi di competitività...



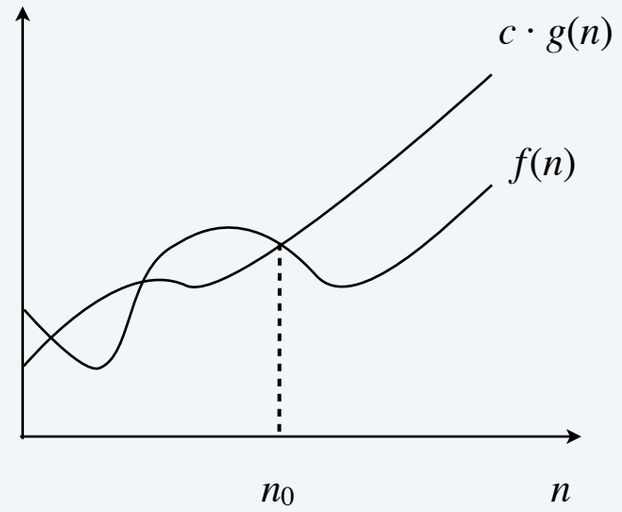
SECTION 2.2

2. ANALISI DEGLI ALGORITMI

- *trattabilità computazionale*
- *ordine asintotico di crescita*
- *implementazione di Gale-Shapley*
- *tempi di esecuzione più comuni*

Notazione O grande

Maggiorazioni. $f(n)$ è $O(g(n))$ se esistono costanti $c > 0$ e $n_0 \geq 0$ tali che $0 \leq f(n) \leq c \cdot g(n)$ per ogni $n \geq n_0$.



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

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

Uso tipico. Insertion sort utilizza $O(n^2)$ confronti per ordinare n elementi.

Analisi degli algoritmi: quiz 1

Sia $f(n) = 3n^2 + 17\ n \log_2 n + 1000$. Quale delle seguenti è vera?

A. $f(n)$ è $O(n^2)$.

B. $f(n)$ è $O(n^3)$.

C. Sia A che B.

D. Né A né B.

Usi e abusi della notazione O grande

"Uguaglianza" a senso unico. $O(g(n))$ è un insieme di funzioni, ma si scrive di norma $f(n) = O(g(n))$ invece di $f(n) \in O(g(n))$.

Es. Consideriamo $g_1(n) = 5n^3$ e $g_2(n) = 3n^2$.

- Abbiamo $g_1(n) = O(n^3)$ e $g_2(n) = O(n^3)$.
- Ma, non se ne conclude che $g_1(n) = g_2(n)$.

Dominio e codominio. f e g sono funzioni a valori reali.

- Il dominio tipicamente sono i numeri interi: $\mathbb{N} \to \mathbb{R}$.
- Talvolta lo estendiamo ai reali: $\mathbb{R}_{\geq 0} \to \mathbb{R}$.
- O lo restringiamo ad un sottoinsieme.

taglia dell'input, relazioni di ricorrenza

grafici, limiti, analisi

Conclusione. È OK abusare la notazione in questi modi; non è OK farne usi errati.

Notazione O grande: proprietà

Riflessività. f è $O(f)$.

Costanti. Se f è $O(g)$ e $c > 0$, allora $c\,f$ è $O(g)$.

Prodotti. Se f_1 è $O(g_1)$ e f_2 è $O(g_2)$, allora $f_1\,f_2$ è $O(g_1\,g_2)$.

Dim.

- $\exists\ c_1 > 0$ e $n_1 \geq 0$ tali che $0 \leq f_1(n) \leq c_1 \cdot g_1(n)$ per ogni $n \geq n_1$.
- $\exists\ c_2 > 0$ e $n_2 \geq 0$ tali che $0 \leq f_2(n) \leq c_2 \cdot g_2(n)$ per ogni $n \geq n_2$.
- Quindi, $0 \leq f_1(n) \cdot f_2(n) \leq \underbrace{c_1 \cdot c_2}_{c} \cdot g_1(n) \cdot g_2(n)$ per ogni $n \geq \underbrace{\max\{n_1, n_2\}}_{n_0}$. ▪

Somme. Se f_1 è $O(g_1)$ e f_2 è $O(g_2)$, allora $f_1 + f_2$ è $O(\max\{g_1, g_2\})$.

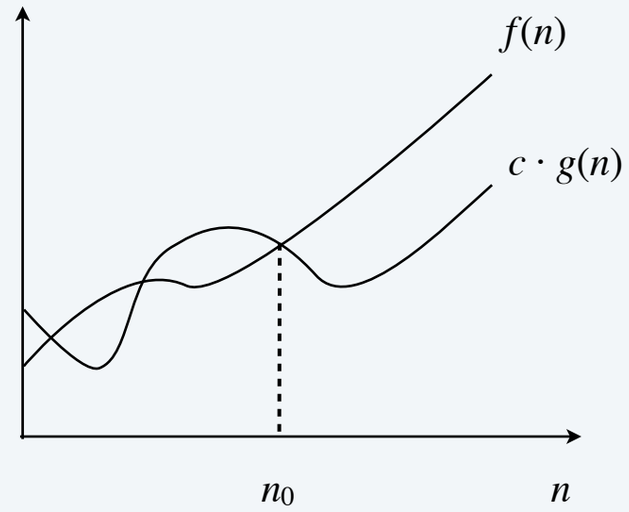
ignora termini di ordine inferiore

Transitività. Se f è $O(g)$ e g è $O(h)$, allora f è $O(h)$.

Es. $f(n) = 5n^3 + 3n^2 + n + 1234$ è $O(n^3)$.

Notazione Omega

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



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

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

Uso tipico. Ogni algoritmo di ordinamento basato su confronti richiede $\Omega(n \log n)$ confronti nel caso peggiore.

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

Analisi degli algoritmi: quiz 2'

Qual è una definizione equivalente della notazione Omega?

A. $f(n)$ è $\Omega(g(n))$ sse $g(n)$ è $O(f(n))$.

B. $f(n)$ è $\Omega(g(n))$ sse esiste una costante $c > 0$ tale che $f(n) \geq c \cdot g(n) \geq 0$ per infiniti n.

C. Sia A che B.

D. Né A né B.

Analisi degli algoritmi: quiz 2'

Qual è una definizione equivalente della notazione Omega?

A. $f(n)$ è $\Omega(g(n))$ sse $g(n)$ è $O(f(n))$.

B. $f(n)$ è $\Omega(g(n))$ sse esiste una costante $c > 0$ tale che $f(n) \geq c \cdot g(n) \geq 0$ per infiniti n.

← perché non usare questa definizione?

C. Sia A che B.

D. Né A né B.

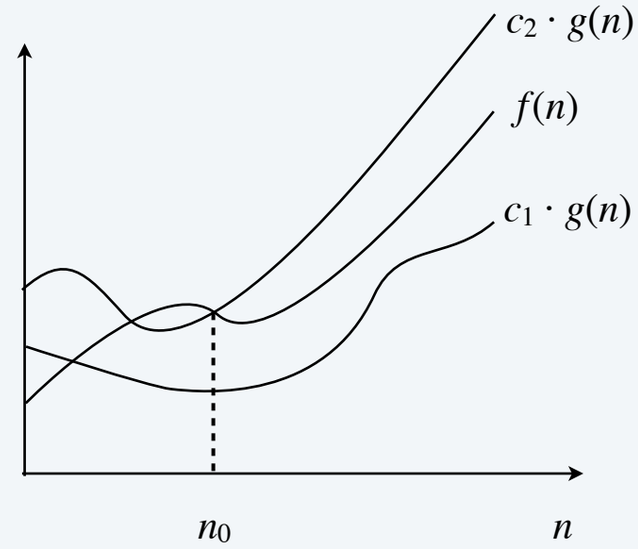
$f(n)$ è $\Omega(g(n))$ se esistono $c_1 > 0$ e $n_0 \geq 0$ tali che $f(n) \geq c_1 \cdot g(n) \geq 0$ per ogni $n \geq n_0$

$g(n)$ è $O(f(n))$ se esistono $c_2 > 0$ e $n_0 \geq 0$ tali che $0 \leq g(n) \leq c_2 \cdot f(n)$ per ogni $n \geq n_0$

$$c_1 = 1 / c_2$$

Notazione Theta grande

Stime strette. $f(n)$ è $\Theta(g(n))$ se esistono costanti $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ tali che $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ per ogni $n \geq n_0$.



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

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

Uso tipico. Mergesort effettua $\Theta(n \log n)$ confronti per ordinare n elements.

tra ½ $n \log_2 n$
e $n \log_2 n$

Analisi degli algoritmi: quiz 3

Qual è una definizione equivalente della notazione Theta grande?

A. $f(n)$ è $\Theta(g(n))$ sse $f(n)$ è sia $O(g(n))$ che $\Omega(g(n))$.

B. $f(n)$ è $\Theta(g(n))$ sse $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ per qualche costante $0 < c < \infty$.

C. Sia A che B.

D. Né A né B.

Limitazioni asintotiche e limiti

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

Dim.

- Per definizione del limite, per ogni $\varepsilon > 0$, esiste n_0 tale che

$$c - \epsilon \leq \frac{f(n)}{g(n)} \leq c + \epsilon$$

per ogni $n \geq n_0$.

- Scegliamo $\varepsilon = \frac{1}{2}\, c > 0$.
- Moltiplicando per $g(n)$: $1/2\, c \cdot g(n) \leq f(n) \leq 3/2\, c \cdot g(n)$ per ogni $n \geq n_0$.
- Quindi, $f(n)$ è $\Theta(g(n))$ per definizione, con $c_1 = 1/2\, c$ e $c_2 = 3/2\, c$. ▪

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

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

Limitazione asintotiche delle funzioni più comuni

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

Dim.

$$\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$$

Logaritmi. $\log_a n$ è $\Theta(\log_b n)$ per ogni $a > 1$ ed ogni $b > 1$.

Dim. $\dfrac{\log_a n}{\log_b n} = \dfrac{1}{\log_b a}$

non serve specificare la base (assunta costante)

Logaritmi e polinomi. $\log_a n$ è $O(n^d)$ per ogni $a > 1$ ed ogni $d > 0$.

Dim.

$$\lim_{n \to \infty} \frac{\log_a n}{n^d} = 0$$

Esponenziali e polinomi. n^d è $O(r^n)$ per ogni $r > 1$ ed ogni $d > 0$.

Dim.

$$\lim_{n \to \infty} \frac{n^d}{r^n} = 0$$

Fattoriali. $n!$ è $2^{\Theta(n \log n)}$.

Dim. Formula di Stirling: $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

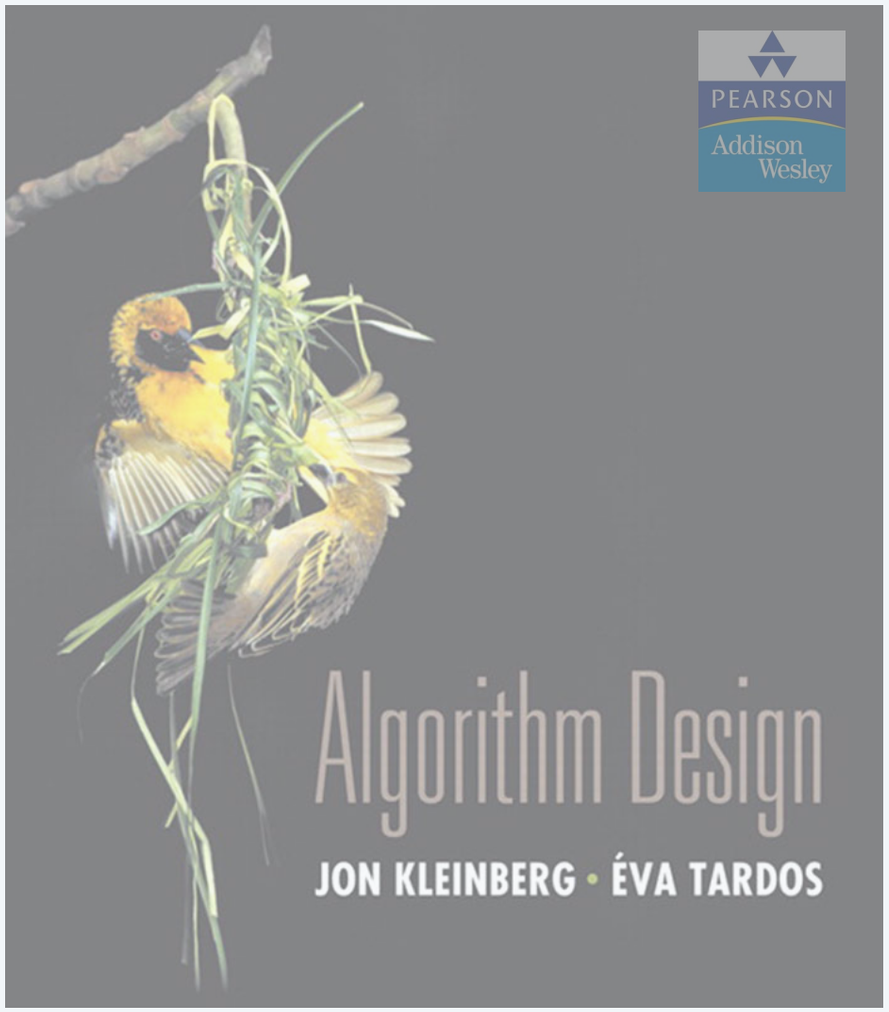
Notazione O grande nel caso di più variabili

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

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

- $f(m, n)$ è sia $O(mn^2 + n^3)$ che $O(mn^3)$.
- $f(m, n)$ è $O(n^3)$ se una precondizione al problema implica $m \leq n$.
- $f(m, n)$ non è né $O(n^3)$ né $O(mn^2)$.

Uso tipico. Nel caso peggiore, la ricerca in ampiezza richiede tempo
$O(m + n)$ per trovare un cammino minimo da s a t in un digrafo con n nodi ed m archi.



SECTION 2.3

2. ANALISI DEGLI ALGORITMI

- *trattabilità computazionale*
- *ordine asintotico di crescita*
- *implementare Gale-Shapley*
- *tempi di esecuzione più comuni*

Implementazione efficiente

Scopo. Implementare Gale–Shapley in modo che usi tempo $O(n^2)$.

GALE–SHAPLEY (*liste di preferenze per n ospedali e n studenti*)

INIZIALIZZA M come abbinamento vuoto.

WHILE (qualche ospedale h è non abbinato)

  $s \leftarrow$ primo studente sulla lista di h cui h non si è già proposto.

  IF (s è non abbinato)

    Aggiungi h–s all'abbinamento M.

  ELSE IF (s preferisce h al partner corrente h')

    Rimpiazza h'–s con h–s nell'abbinamento M.

  ELSE

    s rifiuta h.

RETURN abbinamento stabile M.

Implementazione efficiente

Scopo. Implementare Gale–Shapley in modo che usi tempo $O(n^2)$.

Rappresentazione di ospedali e studenti. Indicizziamo sia ospedali che studenti con $1, \ldots, n$.

Rappresentazione dell'abbinamento.

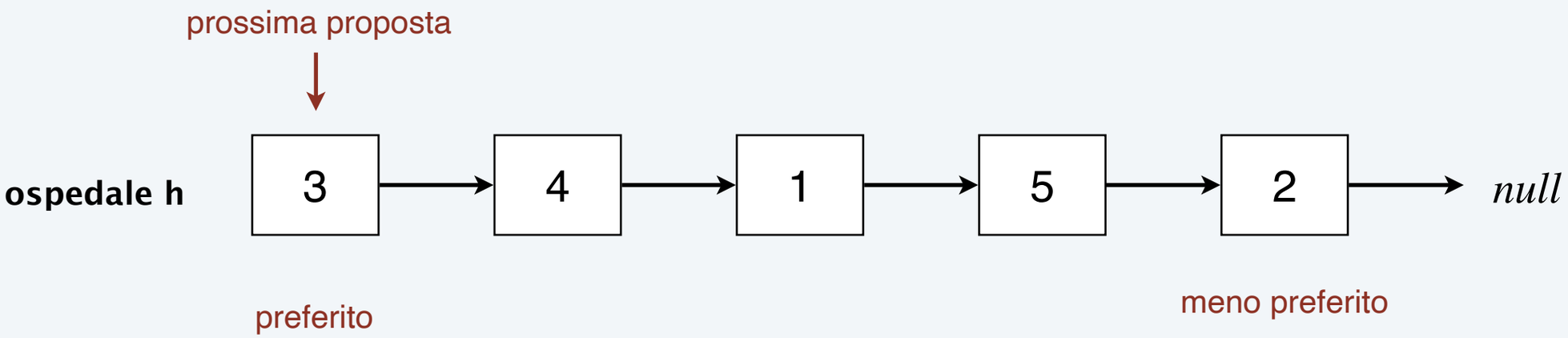
- Manteniamo due array $student[h]$ e $hospital[s]$.
  - se h è abbinato ad s, allora $student[h] = s$ e $hospital[s] = h$
  - il valore 0 designa un ospedale o studente non abbinato
- Per aggiungere/rimuovere una coppia dal matching basta tempo $O(1)$.

- Teniamo l'insieme degli ospedali non abbinati in una pila (o coda).
- Un ospedale non abbinato può essere trovato in tempo $O(1)$.

Rappresentazione dei dati: effettuare una proposta

Proposte da parte degli ospedali.

- Operazione chiave: trovare il successivo studente preferito dall'ospedale.
- Per ogni ospedale: teniamo una lista di studenti, ordinata per preferenza.
- Per ogni ospedale: teniamo un puntatore al prossimo studente cui proporsi.



Bottom line. Making a proposal takes $O(1)$ time.

Rappresentazione dati: accettare/rifiutare una proposta

Accettazione/rifiuto da parte degli studenti.

- Lo studente s preferisce l'ospedale h all'ospedale h' ?
- Per ogni studente, creiamo l'inversa della sua lista di preferenze.

pref[]

1st	2nd	3rd	4th	5th	6th	7th	8th
8	3	7	1	4	5	6	2

lo studente preferisce l'ospedale 4 al 6

poiché rank[4] < rank[6]

rank[]

1	2	3	4	5	6	7	8
4th	8th	2nd	5th	6th	7th	3rd	1st

```
for i = 1 to n
    rank[pref[i]] = i
```

Conseguenza. Dopo un tempo di preprocessamento $\Theta(n^2)$ (per creare gli n array di ranking), basta tempo $O(1)$ per accettare/rifiutare una proposta.

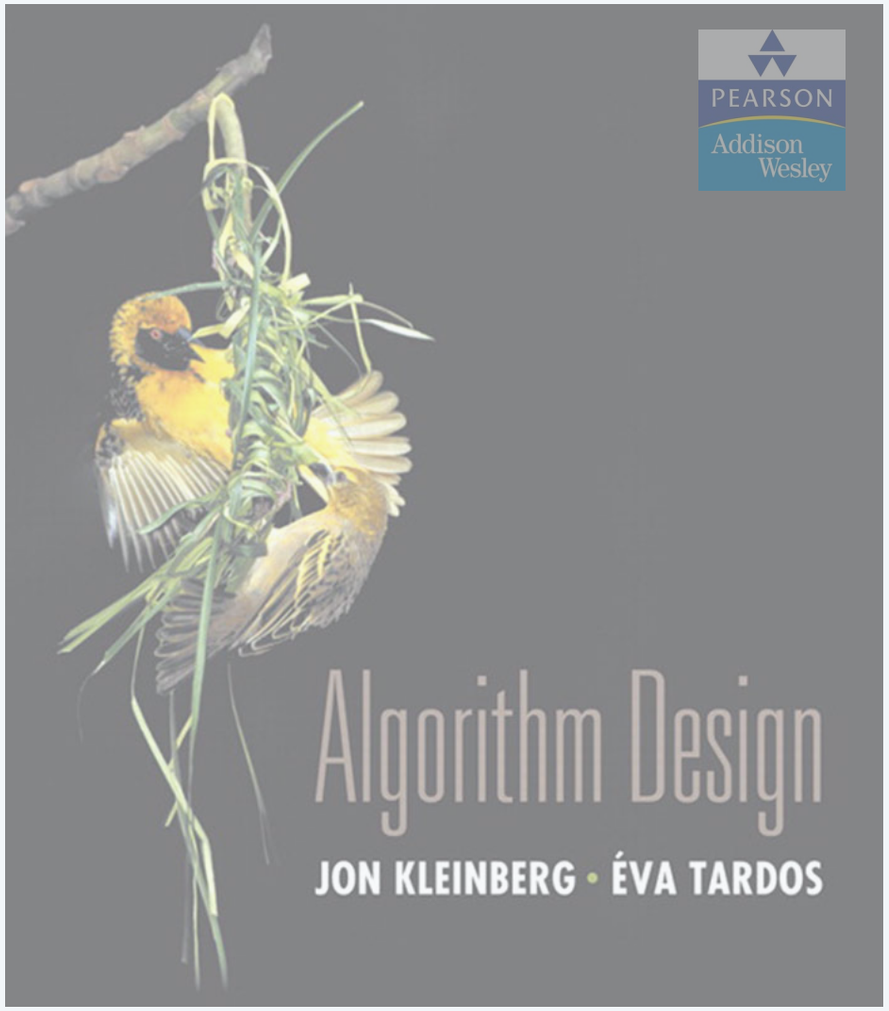
Abbinamento stabile: riepilogo

Teorema. Si può implementare Gale–Shapley in modo che usi tempo $O(n^2)$.

Dim.

- Tempo di preprocessamento $\Theta(n^2)$ per creare gli n array di ranking.
- Ci sono $O(n^2)$ proposte; ciascuna di queste può essere elaborata in tempo $O(1)$.
- ∎

Teorema. Nel caso peggiore, qualunque algoritmo per determinare un abbinamento stabile deve accedere alle liste di preferenze degli ospedali $\Omega(n^2)$ volte.



SECTION 2.4

2. ANALISI DEGLI ALGORITMI

- *trattabilità computazionale*
- *ordine asintotico di crescita*
- *implementare Gale–Shapley*
- *tempi di esecuzione più comuni*

Tempo costante

Tempo costante. Tempo di esecuzione $O(1)$.

maggiorato da una costante,
indipendente dalla taglia dell'input n

Esempi.

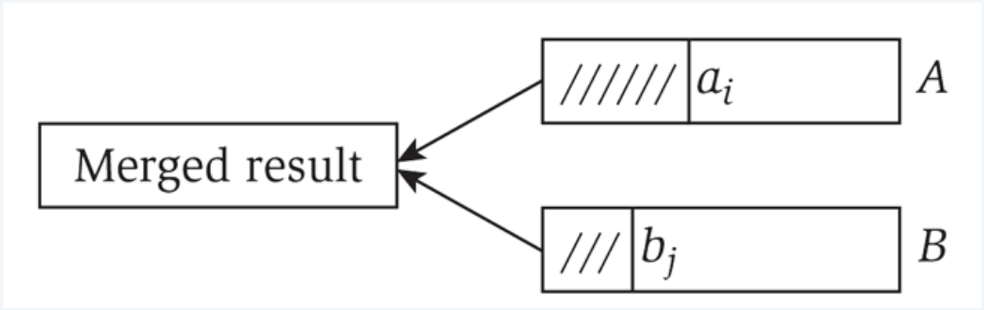
- Costrutto condizionale.
- Operazione logico/aritmetica.
- Dichiarazione/inizializzazione di variabile (di un tipo primitivo).
- Seguire un puntatore in una lista collegata.
- Accesso all'elemento i in un array.
- Confronto/scambio di due elementi in un array.
- …

Tempo lineare

Tempo lineare. Tempo di esecuzione $O(n)$.

Fusione di due liste ordinate. Combinare due liste ordinate $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_n$ in un'unica lista ordinata.

Algoritmo $O(n)$. Passo di merge nel mergesort.



$i \leftarrow 1$; $j \leftarrow 1$.

WHILE (entrambe le liste sono non vuote)

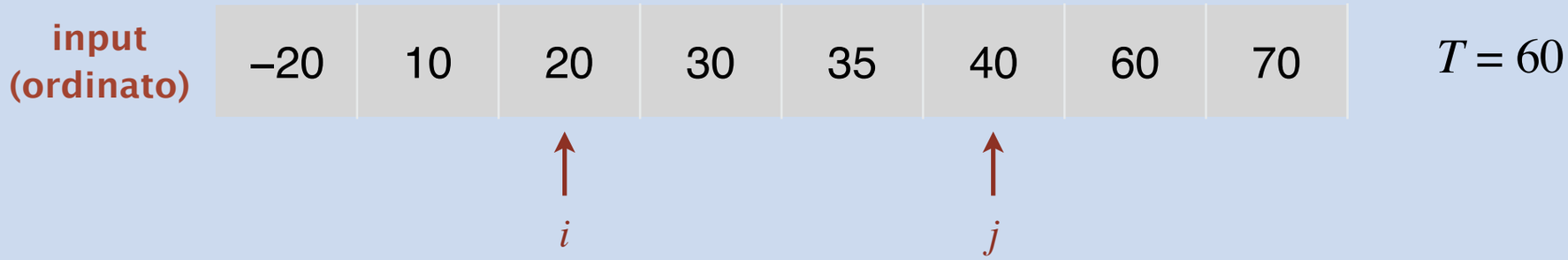
  IF ($a_i \leq b_j$) apponi a_i alla lista di output e incrementa i.

  ELSE apponi b_j alla lista di output e incrementa j.

Apponi gli elementi rimanenti dalla lista non vuota all'output.

TARGET SUM

TARGET-SUM. Dato un array ordinato di n interi distinti ed un intero T, trovarne due che abbiano somma esattamente T .

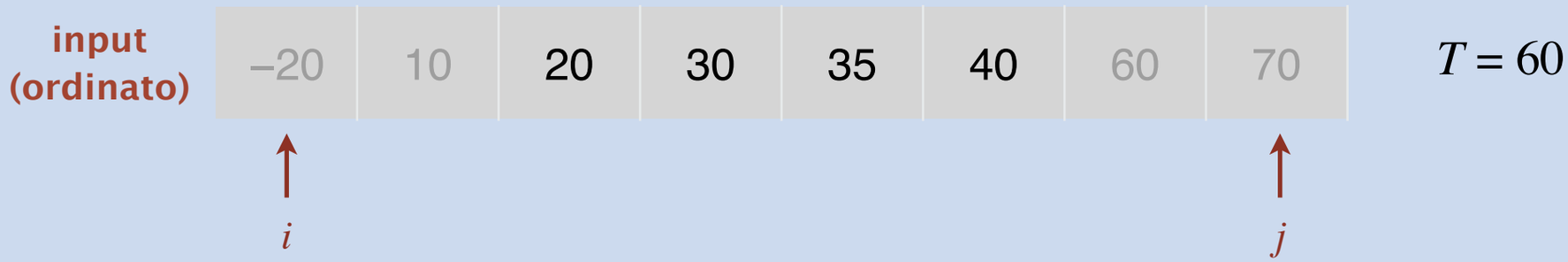


TARGET SUM

TARGET-SUM. Dato un array ordinato di n interi distinti ed un intero T, trovarne due che abbiano somma esattamente T .

Algoritmo $O(n^2)$. Provare tutte le coppie.

Algoritmo $O(n)$. Sfruttare la proprietà di ordinamento.



Invariante. Nessun elemento a sinistra di i o a destra di j è parte di una coppia che ha somma T.

Tempo logaritmico

Tempo logaritmico. Tempo di esecuzione $O(\log n)$.

Ricerca in un array ordinato. Dato un array ordinato A di n interi distinti ed un intero x, trovare un elemento a valore x nell'array (se esiste).

Algoritmo $O(\log n)$. Ricerca binaria.

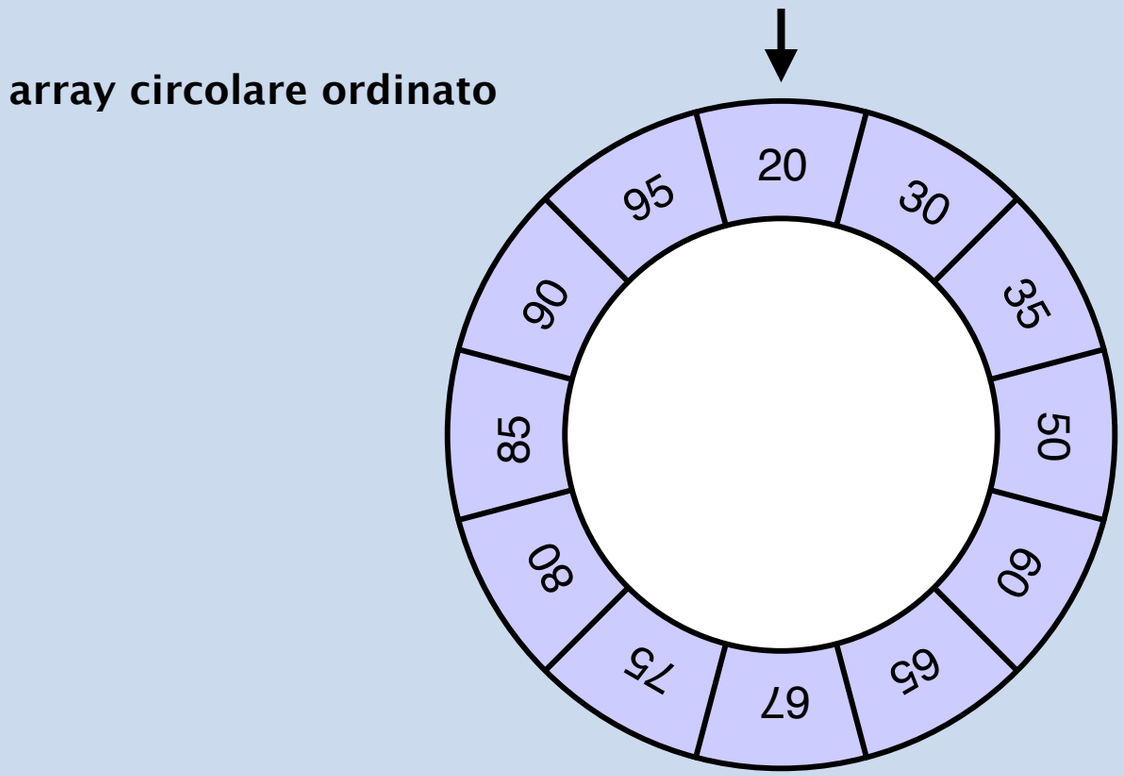
elementi rimanenti

- Invariante: Se x è nell'array, allora x è in $A[lo \,..\, hi]$.
- Dopo k iterazioni del ciclo WHILE, $(hi - lo + 1) \le n / 2^k \Rightarrow k \le 1 + \log_2 n$.

```
lo ← 1; hi ← n.
WHILE (lo ≤ hi)
   mid ← ⌊(lo + hi) / 2⌋.
   IF      (x < A[mid]) hi ← mid − 1.
   ELSE IF (x > A[mid]) lo ← mid + 1.
   ELSE RETURN mid.
RETURN −1.
```

Ricerca in un array ordinato ruotato

Search-In-Sorted-Rotated-Array. Dato un array ordinato ruotato di n interi distinti ed un elemento x, determinare se x è nell'array.



array ordinato ruotato

80	85	90	95	20	30	35	50	60	65	67	75
1	2	3	4	5	6	7	8	9	10	11	12

Ricerca in un array ordinato ruotato

Search-In-Sorted-Rotated-Array. Dato un array ordinato ruotato di n interi distinti ed un elemento x, determinare se x è nell'array.

Algoritmo $O(\log n)$.

- Trovare l'indice k dell'elemento minimo.
- Ricerca binaria di x in $A[1 \,..\, k\text{–}1]$ e in $A[k \,..\, n]$.

trovare l'indice dell'elemento minimo

$lo \leftarrow 1; hi \leftarrow n.$

IF $(A[lo] \leq A[hi])$ RETURN 0 ⟵ non è ruotato

WHILE $(lo + 2 \leq hi)$ ⟵ almeno 3 elementi

  $mid \leftarrow \lfloor (lo + hi) / 2 \rfloor.$

  IF $(A[mid] < A[hi])$ $hi \leftarrow mid.$

  ELSE IF $(A[mid] > A[hi])$ $lo \leftarrow mid.$

RETURN hi

invariante: $A[lo] > A[hi]$

Tempo linearitmico

Tempo linearitmico. Tempo di esecuzione $O(n \log n)$.

Ordinamento. Dato un array di n elementi, ridisporli in ordine crescente.

Algoritmo $O(n \log n)$. Mergesort.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Massimo Intervallo Vuoto

Largest-Empty-Interval. Dati n orari $x_1, \ldots, x_n$ durante i quali delle copie di un file raggiungono un server, qual è il più grande intervallo di tempo in cui il server non è raggiunto da nessuna copia?

Massimo Intervallo Vuoto

Largest-Empty-Interval. Dati n orari $x_1, \ldots, x_n$ durante i quali delle copie di un file raggiungono un server, qual è il più grande intervallo di tempo in cui il server non è raggiunto da nessuna copia?

Algoritmo $O(n \log n)$.

- Ordinare l'array x.
- Scandire la lista ordinata, identificando il buco massimo tra orari successivi.

Tempo quadratico

Tempo quadratico. Tempo di esecuzione $O(n^2)$.

Coppia di punti più vicini. Data una lista di n punti nel piano $(x_1, y_1), \ldots, (x_n, y_n)$, trovare la coppia di punti più vicini tra loro.

Algoritmo $O(n^2)$. Enumerare tutte le coppie di punti (con $i < j$).

```
min ← ∞.
FOR i = 1 TO n
    FOR j = i + 1 TO n
        d ← (x_i − x_j)^2 + (y_i − y_j)^2.
        IF (d < min)
            min ← d.
```

Nota. $\Omega(n^2)$ sembra inevitabile, ma è solo un'apparenza. [vedi §5.4]

Tempo cubico

Tempo cubico. Tempo di esecuzione $O(n^3)$.

3-SUM. Dato un array di n interi distinti, trovarne tre che sommino a 0.

Algoritmo $O(n^3)$. Enumerare tutte le triple (con $i < j < k$).

```
FOR i = 1 TO n
    FOR j = i + 1 TO n
        FOR k = j + 1 TO n
        IF (a_i + a_j + a_k = 0)
            RETURN (a_i, a_j, a_k).
```

Note. $\Omega(n^3)$ sembra inevitabile, ma ottenere $O(n^2)$ non è difficile. [vedi prossima slide]

3-SUM

3-SUM. Dato un array di n interi distinti, trovarne tre che sommino a 0.

Algoritmo $O(n^3)$. Provare tutte le triple.

Algoritmo $O(n^2)$.

3-SUM

3-SUM. Dato un array di n interi distinti, trovarne tre che sommino a 0.

Algoritmo $O(n^3)$. Provare tutte le triple.

Algoritmo $O(n^2)$.

- Ordinare l'array a.
- Per ogni intero a_i : risolvere TARGET-SUM sull'array contenente tutti gli elementi eccetto a_i con somma bersaglio $T = -a_i$.

Algoritmo migliore noto. $O(n^2 / (\log n / \log \log n))$.

Congettura. Non esiste un algoritmo $O(n^{2-\varepsilon})$ per nessun $\varepsilon > 0$.

Tempo polinomiale

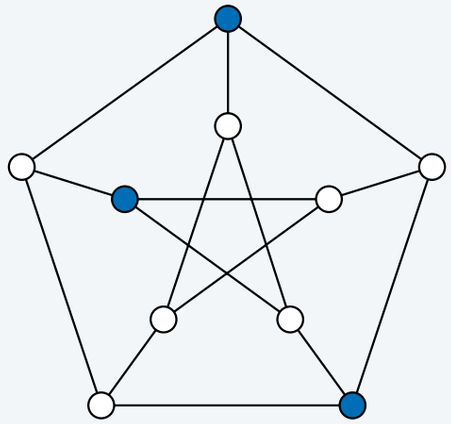
Tempo polinomiale. Tempo di esecuzione $O(n^k)$ per qualche costante $k > 0$.

Insieme indipendente di taglia k. Dato un grafo, trovare k nodi tali che nessuna coppia di essi sia adiacente.

k è una costante

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

FOREACH sottoinsieme S di k nodi:
  Verifica se S è un insieme indipendente.
  IF (S è un insieme indipendente)
    RETURN S.



insieme indipendente di taglia 3

- Verificare se S è un insieme indipendente di taglia k prende tempo $O(k^2)$.
- Il numero di sottoinsiemi di k-elementi è $\binom{n}{k} = \frac{n(n-1)(n-2) \times \cdots \times (n-k+1)}{k(k-1)(k-2) \times \cdots \times 1} \leq \frac{n^k}{k!}$
- $O(k^2 \, n^k / k!) = O(n^k)$.

tempo polinomiale per (esempio) $k = 17$, ma non è pratico

Tempo esponenziale

Tempo esponenziale. Tempo di esecuzione $O(2^{n^k})$ per qualche costante $k > 0$.

Insieme indipendente. Dato un grafo, trovare un insieme indipendente di taglia massima.

Algoritmo $O(n^2\, 2^n)$. Enumerare tutti i sottoinsiemi degli n elementi.

$S^* \leftarrow \emptyset$.

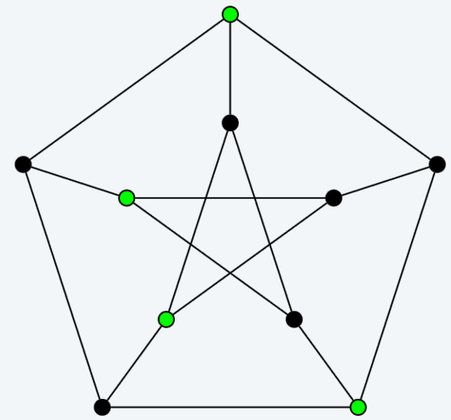
FOREACH sottoinsieme S degli n nodi:

  Verifica se S è un insieme indipendente.

  IF (S è indipendente e $|S| > |S^*|$)

    $S^* \leftarrow S$.

RETURN S^*.



insieme indipendente di taglia massima

Tempo esponenziale

Tempo esponenziale. Tempo di esecuzione $O(2^{n^k})$ per qualche costante $k > 0$.

TSP Euclideo. Dati n punti nel piano, trovare un tour di lunghezza minima.

Algoritmo $O(n \times n!)$. Enumerare tutte le permutazioni di lunghezza n.

$\pi^* \leftarrow \emptyset$.

FOREACH permutazione ciclica π di n punti:

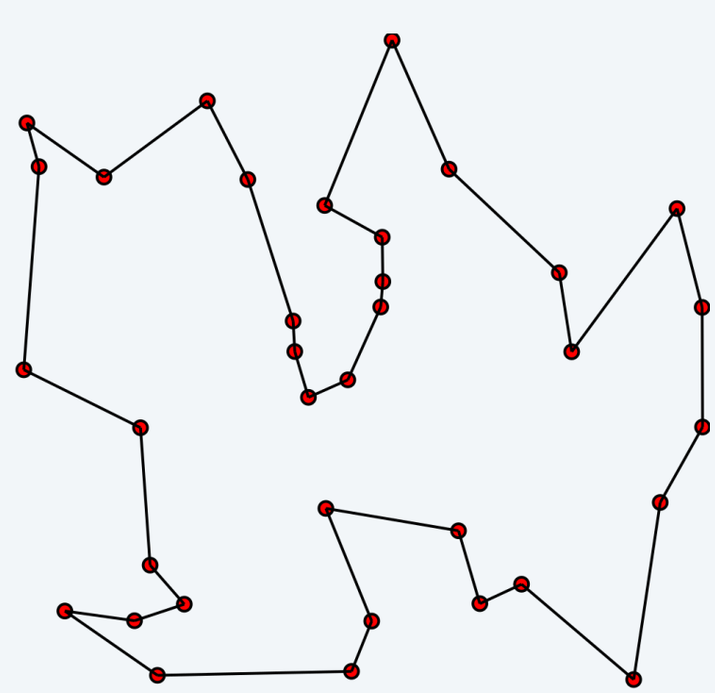
  Calcola la lunghezza del tour corrispondente a π.

  IF (lunghezza(π) < lunghezza(π^*))

    $\pi^* \leftarrow \pi$.

RETURN π^*.

per semplicità, assumiamo che le distanze euclidee siano arrotondate all'intero più vicino (per evitare problematiche di precisione infinita)



Analisi degli algoritmi: quiz 4

Qual è una definizione equivalente di tempo esponenziale?

A. $O(2^n)$

B. $O(2^{cn})$ per qualche costante $c > 0$

C. Sia A che B.

D. Né A né B.