

Strutture Dati e Algoritmi

Trattabilità e complessità computazionale

Luca Di Gaspero, Università degli Studi di Udine

Problemi decidibili e indecidibili

Classificazione dei problemi

Non tutti i problemi computazionali ammettono algoritmi di risoluzione: ad esempio il *problema della fermata (halting problem)* (Turing, 1937)

Dato un generico algoritmo (o programma) A in ingresso, esso termina o va in ciclo infinito?

Non c'è una soluzione al problema nel caso generale (ossia una dimostrazione che consenta di rispondere per un *qualunque algoritmo*)

In alcuni casi specifici tuttavia è possibile dare delle risposte

Problemi decidibili

- Un problema è detto **decidibile** (o **calcolabile**) se esiste un algoritmo che **per ogni sua istanza** sia in grado di **terminare** la sua esecuzione pervenendo ad una soluzione
- Ad esempio, *determinare se un numero è primo*

```
#include <stdbool.h>

bool primo(int n) {
    int fattore;
    if (n == 1)
        return false;
    if (n == 2)
        return true;
    for (fattore = 2; fattore < n / 2; fattore++)
        if (n % fattore == 0)
            return false;
    return true;
}
```

Problemi decidibili

L'algoritmo `primo()` termina su ogni sua istanza (ovvero per qualunque valore di `n`)?

- **Si**, sempre: perché se `n` è uno o due la funzione risponde immediatamente, altrimenti la variabile `fattore` viene sempre incrementata di 1 e a un certo punto deve verificare la condizione (almeno quando `fattore` è esattamente uguale a `n / 2`)

Problemi decidibili, algoritmi corretti

L'algoritmo `primo()` è sempre corretto (ovvero restituisce il valore corretto per qualunque valore di `n`)?

- **Si**, o meglio l'ipotesi per cui è corretto è che i numeri negativi non siano considerati come primi, cosa che è coerente con la definizione:

Problemi indecidibili

- Un problema è **indecidibile** (o **non calcolabile**) se nessun algoritmo sia in grado di terminare la sua esecuzione pervenendo ad una soluzione su **ogni sua istanza**

Dato un programma **A**, non esiste un algoritmo per stabilire se esso terminerà o meno la sua esecuzione su un qualunque input

Problema della fermata (Turing, 1937)

Dimostrazione di indecidibilità del problema della fermata

Assunzioni (⚠ *don't try this at home*):

1. La sequenza di simboli (caratteri) che costituisce il programma `A` può essere interpretata sia come dato che come programma (le stringhe che costituiscono il programma stesso)
2. Un programma può essere dato in pasto a un altro programma (ad esempio ad un compilatore, ma anche più semplicemente le funzioni che accettano altre funzioni), ovvero è lecito scrivere qualcosa come `C(A)` con `C` un altro programma

Dimostrazione di indecidibilità del problema della fermata

Supponiamo ora, **per assurdo**, che esista un algoritmo `termina(A, D)` che, *in tempo finito* valuta, *senza eseguirlo*[†], quello che accadrebbe con l'esecuzione di `A(D)` ;

- in particolare `termina` restituirebbe `true` se l'algoritmo `A(D)` terminasse e `false` se `A(D)` andasse in un ciclo infinito

Osserviamo che le assunzioni 1 e 2 descritte in precedenza implicano che è lecito e possibile invocare `termina(A, A)`

[†] se lo eseguisse e `A` andasse in ciclo infinito ovviamente richiederebbe tempo infinito per restituire `false`

Indecidibilità del problema della fermata

Costruiamo ora il seguente programma:

```
paradosso(A) {  
    while (termina(A, A))  
        ;  
}
```

Il programma `paradosso(A)` è costruito in modo che non termini quando `A` termina sull'input `A` (e viceversa). In altri termini: $\text{termina}(\text{paradosso}, A) \equiv \neg \text{termina}(A, A)$.

Ci chiediamo se `paradosso()` termini su qualunque suo input o, in altri termini, se $\forall D, \text{termina}(\text{paradosso}, D) == \text{true}$?

Indecidibilità del problema della fermata

Consideriamo l'input `D = paradosso`. Esso è lecito per le assunzioni 1 e 2 viste in precedenza.

Valutiamo `termina(paradosso, paradosso)` e verifichiamo l'esito dell'esecuzione.

Ricordando che `termina(paradosso, A) \equiv !termina(A, A)` abbiamo che:

$$\text{termina}(\text{paradosso}, \text{paradosso}) \equiv \text{!termina}(\text{paradosso}, \text{paradosso})$$

Abbiamo dimostrato che non può esistere un algoritmo che possa stabilire se un programma `A` termini o meno su *qualunque* suo input

Indecidibilità del problema della fermata

Anche analizzando il codice che verrebbe eseguito a seguito della chiamata `paradosso(paradosso)` giungiamo alla medesima contraddizione:

`paradosso(paradosso)` \equiv

```
while (termina(paradosso, paradosso))  
    ;
```

Se `termina(paradosso, paradosso)` fosse vero, il ciclo `while` non terminerebbe, mentre se `termina(paradosso, paradosso)` fosse falso, il ciclo `while` terminerebbe.

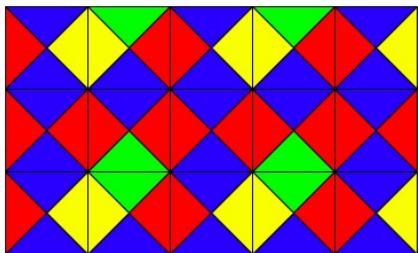
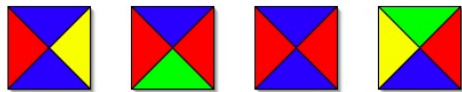
In entrambi i casi l'equivalenza non è coerente

Indecidibilità

- Il problema della fermata è **indecidibile**, ossia non esiste alcun algoritmo di risoluzione che termina su ogni suo input restituendo un risultato
- In realtà il problema è **semidecidibile**, ovvero è facile decidere se termina (quando termina): basta eseguirlo; è impossibile decidere se non termina

Altri problemi indecidibili

- stabilire l'**equivalenza tra due programmi** f, g : per ogni possibile input, producono lo stesso output?
- **tiling problem**: data una scacchiera $n \times m$ e delle tessere di dimensione unitaria, colorate di tipo diverso (ognuna con quattro triangoli colorati in un certo modo), è possibile o meno coprire la scacchiera rispettando il vincolo che piastrelle adiacenti siano dello stesso colore?

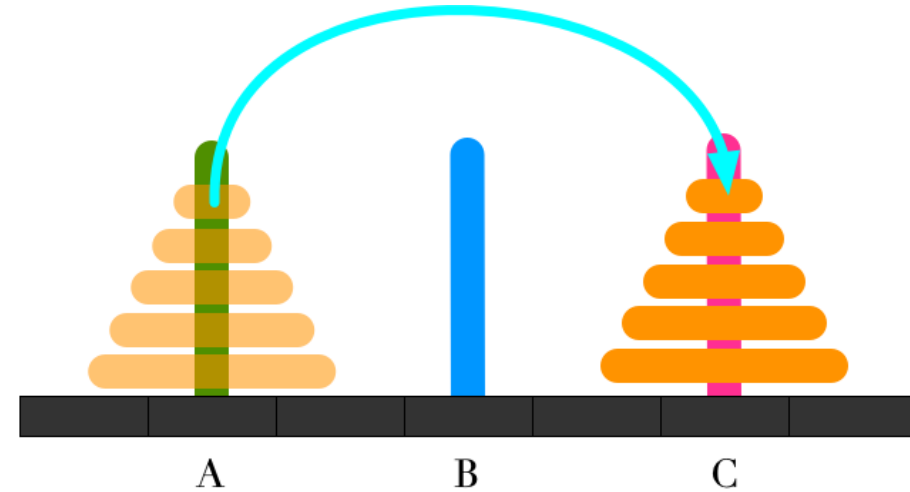


Ovviamente in casi particolari, come quello di lato, è possibile risolvere il problema ma non è possibile farlo nel caso generale con n e m arbitrari

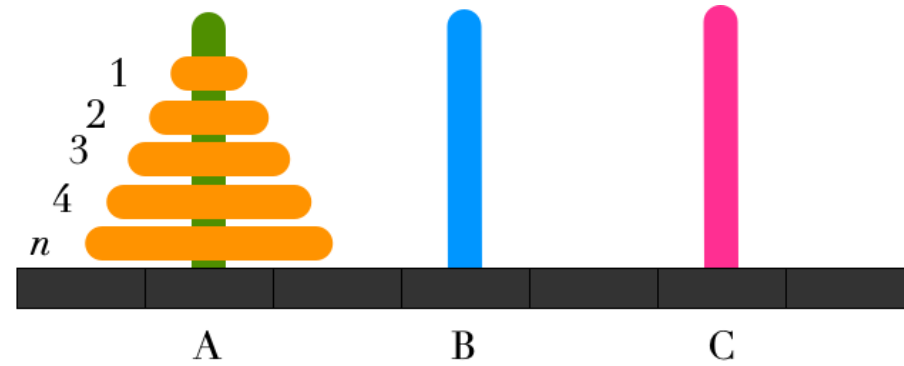
Complessità

Alcuni tipi di problemi, nonostante siano decidibili, possono richiedere tempi di risoluzione elevati: ad esempio il problema delle **Torri di Hanoi**

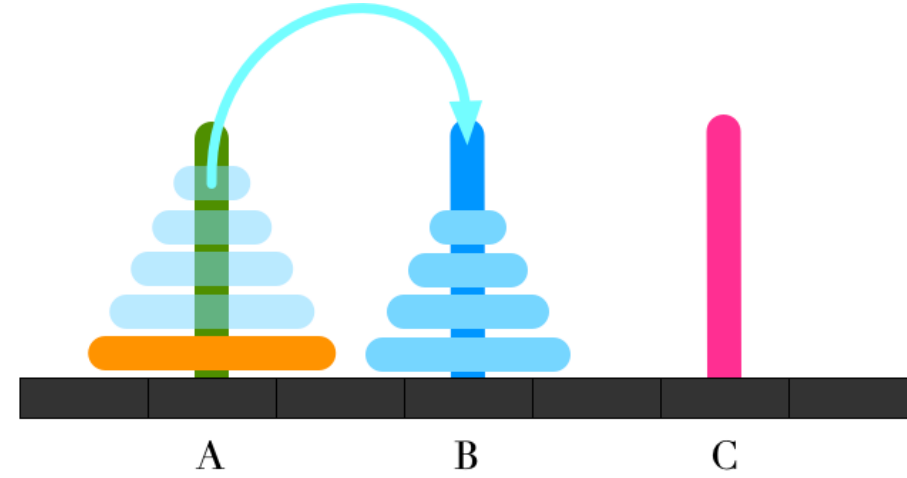
- **Obiettivo:** spostare tutti i dischi dal piolo **A** al piolo **C**
- Ogni mossa sposta un disco in cima a un piolo con il vincolo che un disco non può poggiare su uno più piccolo



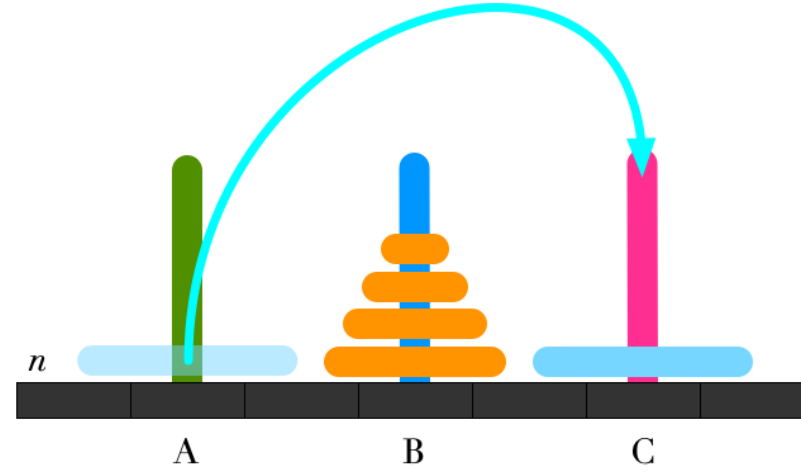
Supponiamo che i pioli siano etichettati con A, B e C, e i dischi numerati da 1 (il più piccolo) a n (il più grande).



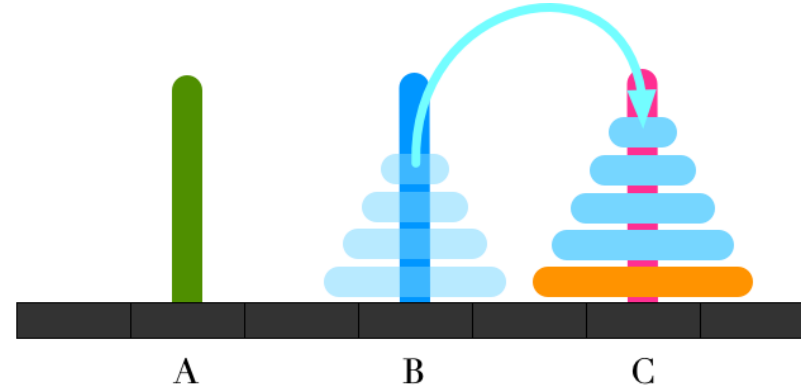
Passo 1: sposta i primi $n - 1$ dischi da A a B



Passo 2: sposta il disco n da A a C



Passo 3: sposta i primi $n - 1$ dischi da B a C



Torri di Hanoi: ricorsione

Per spostare gli n dischi compiamo un'operazione elementare (la 2, spostamento di un disco) e delle operazioni complesse (la 1 e la 3) che, però, hanno la stessa struttura del problema originale, ossia lo spostamento di $n - 1$ dischi, solo su un numero di dischi ridotto di un'unità.

Torri di Hanoi: stampa delle mosse

```
void torri_hanoi(int n, char partenza, char intermedio, char destinazione) {
    if (n > 1) {
        // Sposto tutti gli n - 1 dischi dal piolo partenza
        // al piolo intermedio usando il piolo destinazione come appoggio
        torri_hanoi(n - 1, partenza, destinazione, intermedio);
        /* Sposto il disco rimanente (più largo di tutti) sul piolo destinazione */
        printf("disco %d: %c -> %c\n", n, partenza, destinazione);
        // Sposto tutti gli n - 1 dischi dal piolo intermedio
        // al piolo destinazione usando il piolo partenza come appoggio
        torri_hanoi(n - 1, intermedio, partenza, destinazione);
    } else {
        // È rimasto un solo piolo, lo sposto dal piolo partenza a quello destinazione
        printf("disco %d: %c -> %c\n", n, partenza, destinazione);
    }
}

/* esempio di chiamata iniziale */
torri_hanoi(5, 'A', 'B', 'C');
```

Torri di Hanoi: numero di mosse

Per spostare gli n dischi sono necessarie $2^n - 1$ mosse.

Dimostrabile per induzione:

- **Caso base**, $n = 1$: è necessaria una sola mossa, e coerentemente $1 = 2^1 - 1$
- **Passo induttivo**, $n > 1$: per ipotesi induttiva, lo spostamento di $n - 1$ dischi richiede un numero di passi pari a $2^{n-1} - 1$

L'algoritmo sposta tutti gli $n - 1$ dischi dal piolo di partenza al piolo intermedio, poi sposta il disco rimasto sul piolo di destinazione e infine risposta tutti gli $n - 1$ dischi dal piolo intermedio a quello di destinazione:

$$\begin{array}{ccccccc} \overbrace{(2^{n-1} - 1)}^{n-1 \text{ dischi dal piolo A al B}} & + & \overbrace{1}^{\text{disco } n \text{ a C}} & + & \underbrace{(2^{n-1} - 1)}_{n-1 \text{ dischi dal piolo B al C}} & = & 2(2^{n-1} - 1) + 1 = 2^n - 1 \end{array}$$

Torri di Hanoi: tempo di calcolo

Supponendo di fare *una mossa al secondo*, i tempi di calcolo crescono secondo la seguente legge:

n	5	10	15	20	25	30	35	40	64
tempo	31s	17m	9h	12g	1a	34a	1089a	34865a	$585 \cdot 10^9 a$

Torri di Hanoi: tempo di calcolo

Se invece di fare una mossa al secondo, potessi fare b mosse al secondo per quanti dischi m aggiuntivi rispetto a n riesco a risolvere il gioco in un dato tempo t ?

Devo risolvere l'equazione:

$$(2^{n+m} - 1)/b = t$$

$$2^{n+m} = tb + 1$$

$$n + m = \log_2(tb + 1) \approx \log_2 t + \log_2 b$$

Poiché facendo una mossa al secondo nel tempo t riesco a risolvere n dischi: $n \approx \log_2 t$ e dunque $n + m \approx n + \log_2 b \Rightarrow m \approx \log_2 b$

Torri di Hanoi: tempo di calcolo

Quindi, dato che $m \approx \log_2 b$, se **raddoppio** la mia velocità ($b = 2$) riesco a risolvere 1 solo disco in più.

Ad esempio, per il numero di dischi "trattabili" in un giorno (<24h).

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
$n + m$	17	20	23	26	29	32	35	45

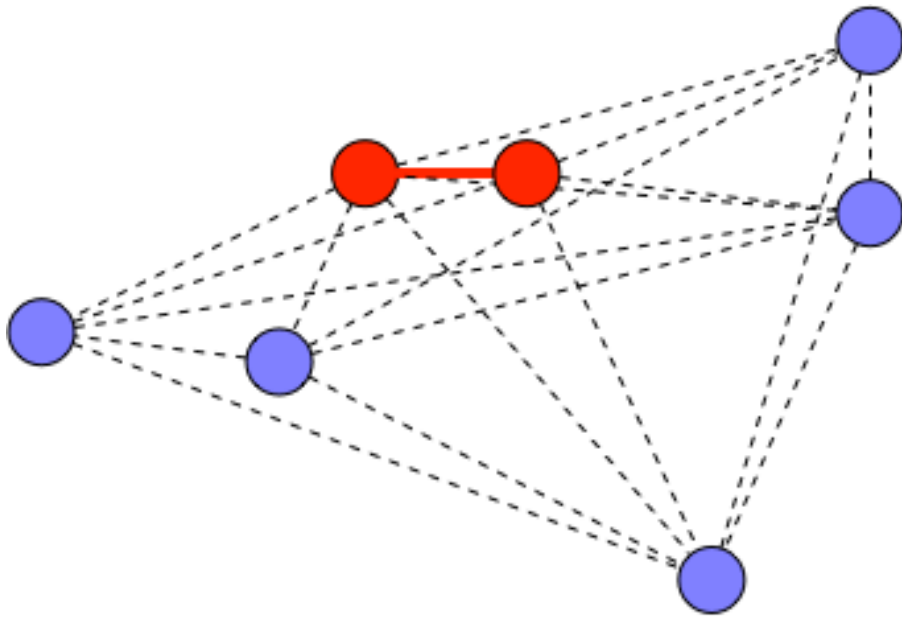
Torri di Hanoi

Il problema delle Torri di Hanoi è **intrattabile**: è dimostrabile che non è possibile usare meno di $2^n - 1$ mosse.

(non può esistere un algoritmo che usi un numero inferiore di mosse)

Problema della ricerca della coppia di punti più vicini

- Dato un insieme di punti P disposti su di un piano, trovare il valore della distanza della coppia di punti che si trova a distanza minima



Problema della ricerca della coppia di punti più vicini

```
# include <math.h>
double punti_piu_vicini(/* insieme di punti */ P, int n) {
    double min = +INFINITY, d;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            d = distanza(P, i, j); // distanza fra i punti i e j
            /* confronto */
            if (i != j && d < min) {
                min = d;
            }
        }
    }
    return min;
}
```

Ricerca della coppia di punti più vicini: tempo di calcolo

Quante operazioni di calcolo della distanza fra punti vengono effettuate?

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n \cdot n = n^2$$

Supponendo di fare un confronto al secondo, i tempi di calcolo crescono secondo la seguente legge:

n	5	10	15	20	25	30	35	40
tempo	25s	100s	4m	7m	10m	15m	20m	26m

Ricerca della coppia di punti più vicini

Se invece di fare un confronto al secondo, si fanno b confronti al secondo nel tempo t riesco a risolvere il problema per un vettore di dimensione $n + m$, dove $t = (n + m)^2 / b$

$$n + m = (tb)^{1/2} = \sqrt{t}\sqrt{b} = n\sqrt{b}$$

Aumentare di un fattore moltiplicativo b (ossia b operazioni/sec) migliora di un fattore moltiplicativo circa pari a \sqrt{b}

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6
$n + m$	64	202	640	2023	6400	20238	64000

Algoritmi esponenziali e algoritmi polinomiali

L'algoritmo delle torri di Hanoi è un algoritmo **esponenziale**: richiede un tempo proporzionale ad una funzione esponenziale della dimensione dell'input

Per inciso, è la stessa situazione delle *Terapie Intensive* in caso di contagi da COVID in assenza di controllo: possiamo raddoppiarle ($b = 2$), ma il numero di giorni che servirà per riempirle interamente si sposta solamente di qualche unità (in questo caso, fortunatamente, la base dell'esponenziale non è 2 ma il problema rimane).

L'algoritmo della ricerca della coppia di punti vicini è un algoritmo **polinomiale** proporzionale a un polinomio della dimensione dell'input (solitamente di grado basso)

Dimensione dei dati per un problema generico

Esistono diverse caratterizzazioni dimensionali per i dati di un problema:

Numero di bit per la rappresentazione dei dati: k bit possono rappresentare interi in $\{0, 1, \dots, 2^k - 1\}$

Esempio, per $k = 3$:

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

Numero di elementi di una struttura dati: array, stringhe, liste, insiemi, alberi

Numero di celle di memoria: occupate dai dati (ognuna contiene $\approx \log n$ bit)

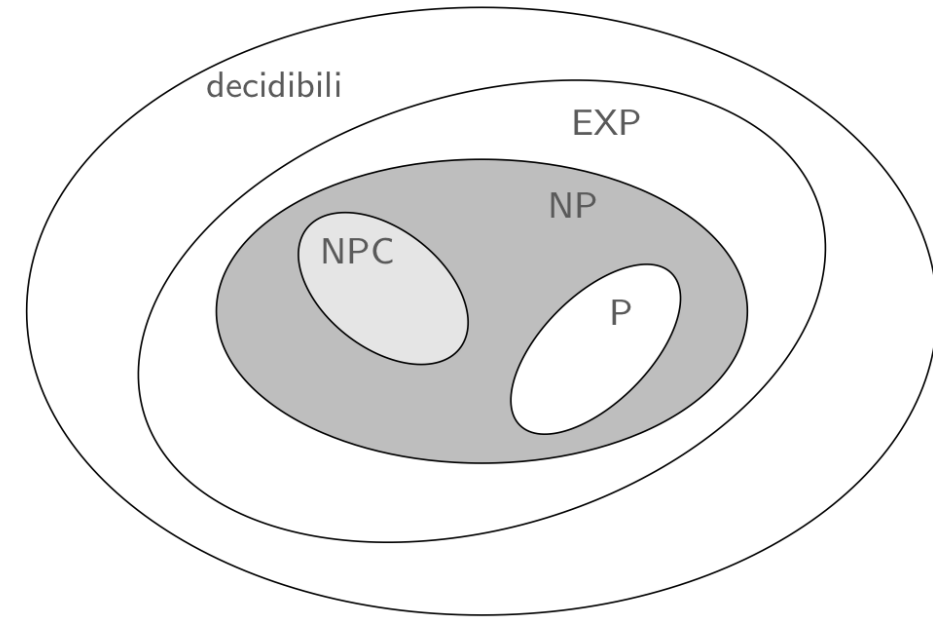
Definizione: Un algoritmo è detto *esponenziale*, nella dimensione del suo input n , se esistono due costanti $c, n_0 > 0$ tali che il numero di passi elementari da esso eseguiti è al più c^n , per ogni input di dimensione n e per ogni $n > n_0$.

Problema *trattabile*: esiste un algoritmo polinomiale che lo risolve

Problema *intrattabile*: non esiste un algoritmo polinomiale che lo risolve

Classi di complessità dei problemi

- P classe dei problemi risolvibili (*deterministicamente*) in tempo polinomiale
- EXP classe dei problemi risolvibili (*deterministicamente*) in tempo esponenziale
- NP classe dei problemi per i quali verificare una soluzione richiede tempo polinomiale (risolvibili (*non-deterministicamente*) in tempo polinomiale)



- NPC classe dei problemi *completi* per NP , detti NP -Completi

Problemi in NP

Basato sul concetto di “*certificato polinomiale*” per un problema computazionale Π

- chi ha la soluzione per un'istanza di Π può convincervi che la soluzione è giusta in tempo polinomiale
- chi non ha tale soluzione deve procedere per tentativi richiedendo tempo esponenziale

Problemi NP

Esempio: date le tre variabili proposizionali $p, q, r \in \{T, F\}$ e la formula logica $\Phi \equiv (p \vee \bar{q} \vee r) \wedge (p \vee r)$, verificare che $p := T, q := F, r := F$ soddisfa la formula Φ è facile, basta sostituire i valori di verità e applicare le regole di valutazione:

$$(T \vee \bar{F} \vee F) \wedge (T \vee F) \equiv T \wedge T \equiv T$$

Trovare la soluzione è necessario provare tutte le 2^n combinazioni $(p, q, r) := (T, T, T), (T, T, F), (T, F, T), \dots$ di valori di verità per p, q, r .

Problemi NP

NP è la classe dei problemi che ammettono un certificato polinomiale

Ovviamente, $P \subseteq NP$




- infatti, qualora il problema Π sia polinomiale (ossia $\Pi \in P$), il certificato coincide con la soluzione del problema

Riguardo la relazione inversa, non sappiamo se $NP \stackrel{?}{\subseteq} P$ oppure $NP \stackrel{?}{\not\subseteq} P$














Esempio di problema NP

Il problema della **battaglia navale** $n \times m$

- Dati un campo di battaglia di dimensioni $n \times m$ in cui alcune celle sono già inizialmente predeterminate (nave/acqua) e una serie di navi di dimensione variabile (e di larghezza 1 unità) posizionare tutte le navi sul campo in modo che:
 1. le indicazioni iniziali relative alle celle predeterminate siano rispettate;
 2. nessuna nave occupi celle (ortogonalmente o diagonalmente) adiacenti;
 3. il numero di celle coperte da una nave nella colonna (riga) i -esima sia uguale ad un valore dato (e predefinito)
- Questo problema è *difficile da risolvere*, ma una sua soluzione è *facile da verificare*

0	1	3	1	
				3
				0
				1
				1

Da piazzare:    e  

0	1	3	1	
				3
				0
				1
				1

È facile verificare che questa è una soluzione, ma è complicato trovarla: (2 orientamenti per ciascuna nave, che possono essere piazzate quasi ovunque, numero esponenziale di possibilità).

Problemi NP -Completi

Un'ulteriore classe di problemi, indicata con NPC è la classe dei cosiddetti problemi NP -completi

- $NPC \subseteq NP$ e, come per i problemi NP non si sa se tali problemi siano trattabili
- Tuttavia, i problemi in NPC hanno una caratteristica fondamentale:

ogni problema $\Pi \in NP$ può essere ricondotto a (*trasformato in*) un problema $\Sigma \in NPC$ attraverso un processo di *riduzione (polinomiale)* ($\Pi \stackrel{P}{\rightsquigarrow} \Sigma$)

Problemi NP -Completi

La questione se $P = NP$ oppure $P \neq NP$ è un famoso problema aperto in informatica

- I problemi in NP -Completi sono la chiave di volta per risolverlo, infatti, se un problema NP -Completo è *trattabile*, allora tutti i problemi in NP lo sono (perché possono essere ridotti a quel problema) e dunque varrebbe $P = NP$
- Viceversa, se un problema NP -Completo è *intrattabile*, allora tutti i problemi in NP lo sono e varrebbe $P \neq NP$
- I problemi NP -Completi sono una sorta di problemi modello per la teoria della complessità, a cui tutti gli altri problemi difficili possono essere ricondotti.

Analisi della complessità

Analisi della complessità

Negli esempi precedenti (Torri di Hanoi, Punti vicini) abbiamo stimato in modo molto grossolano il numero di operazioni principali necessarie alla soluzione del problema su un input di dimensione generica n (spostamento di un disco, calcolo della distanza)

Per avere delle stime più accurate è necessario introdurre un modello di calcolo generico (astratto) e una misura delle operazioni su tale modello di calcolo

Modello di calcolo per l'analisi della complessità

- **Random Access Machine** (RAM: macchina ad accesso diretto)
- Modello di *von Neumann*: dati e programmi sono sequenze (binarie) contenute nella memoria principale
 - contatore di programma (salti) e registro accumulatore (operazioni logiche e aritmetiche)
 - memoria di **dimensione illimitata** e **accesso alla memoria in tempo costante** indipendente dalla posizione
 - processore esegue operazioni aritmetiche, di confronto, logiche, di trasferimento e di controllo
- È un'idealizzazione di un calcolatore reale, con alcune semplificazioni
- **Assunzione**: *costo uniforme delle operazioni*, il costo di una singola istruzione è **costante** e non dipende dalla dimensione dei dati

Analisi di complessità

- Misura di **performance di un algoritmo** espressa in funzione della dimensione dei dati in input n
 - **tempo** numero di operazioni RAM (elementari) eseguite
 - **spazio** numero di celle di memoria occupate (in aggiunta a quelle per l'input)
- **Complessità o costo computazionale** $f(n)$ in *tempo* e *spazio* di un problema Π :
 - *caso pessimo o peggiore*: costo massimo fra tutte le istanze di Π aventi dimensioni dei dati pari a n
 - *caso medio*: costo mediato tra tutte le istanze di Π aventi dimensioni pari a n
 - ~~caso ottimo: costo minimo fra tutte le istanze di Π aventi dimensione dei dati pari a n~~

Analisi di complessità

Scopi:

- stimare il tempo impiegato per elaborare un dato input
- stimare il più grande input gestibile in tempi "ragionevoli"
- **confrontare l'efficienza di algoritmi diversi**
- ottimizzare le parti più importanti

Analisi di complessità

Dimensione dell'input:

- *costo logaritmico*: la taglia dell'input è il numero di bit necessari per rappresentarlo
 - Esempio: moltiplicazione di numeri binari lunghi n bit
- *costo uniforme*: la taglia dell'input è il numero di elementi di cui è costituito
 - Esempio: ricerca minimo in un vettore di n elementi

Possiamo assumere che gli “elementi” siano rappresentati da un numero costante di bit e pertanto le due misure coincidono a meno di una costante moltiplicativa

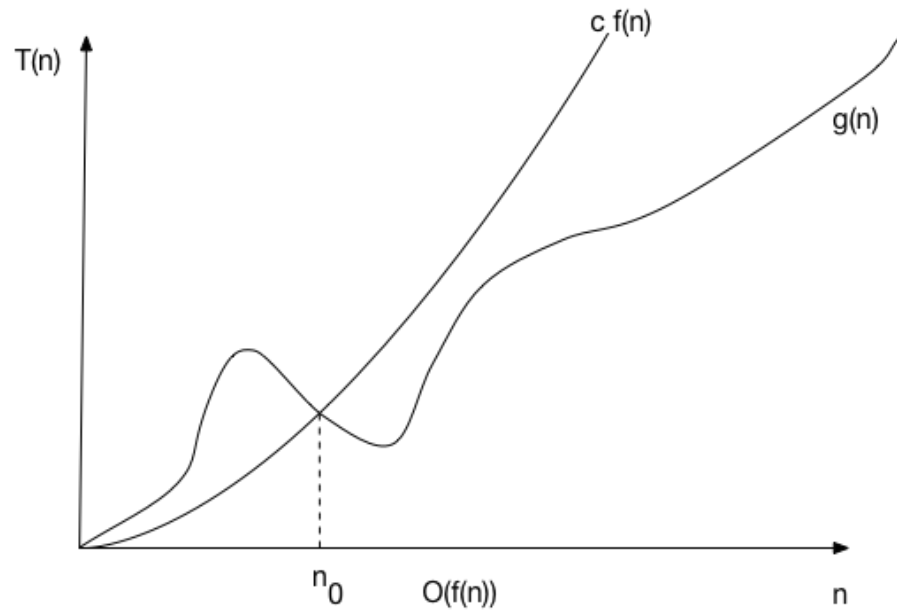
Notazione asintotica e ordini di complessità

Notazione asintotica per la complessità

Non ci interessano risultati troppo accurati, utilizzeremo invece una notazione asintotica, che dà un'idea di come si comporta l'algoritmo al crescere delle dimensioni del suo input

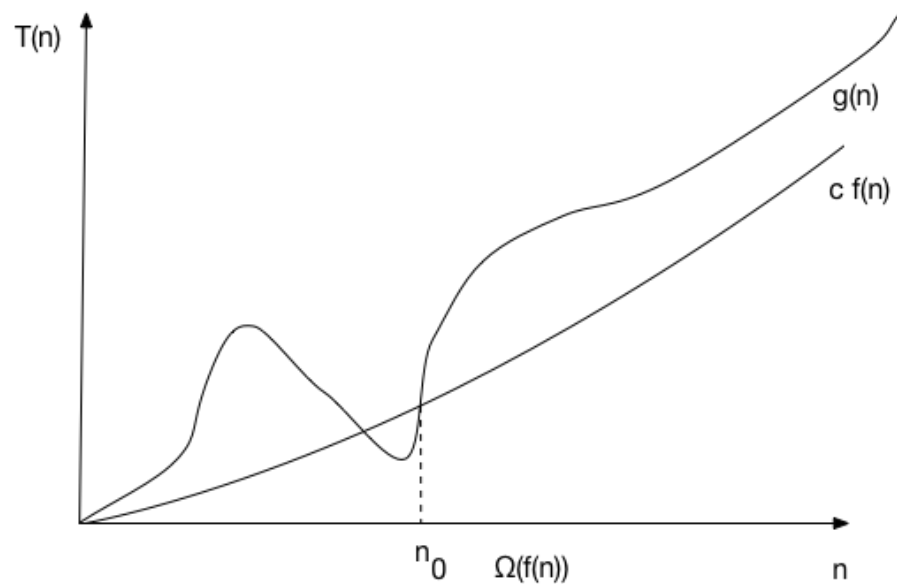
Notazione asintotica per la complessità: limitazione superiore $O(f(n))$

Definiamo $g(n) = O(f(n))$ se e solo se $\exists c, n_0 : g(n) \leq cf(n) \forall n > n_0$



Notazione asintotica per la complessità: limitazione inferiore $\Omega(f(n))$

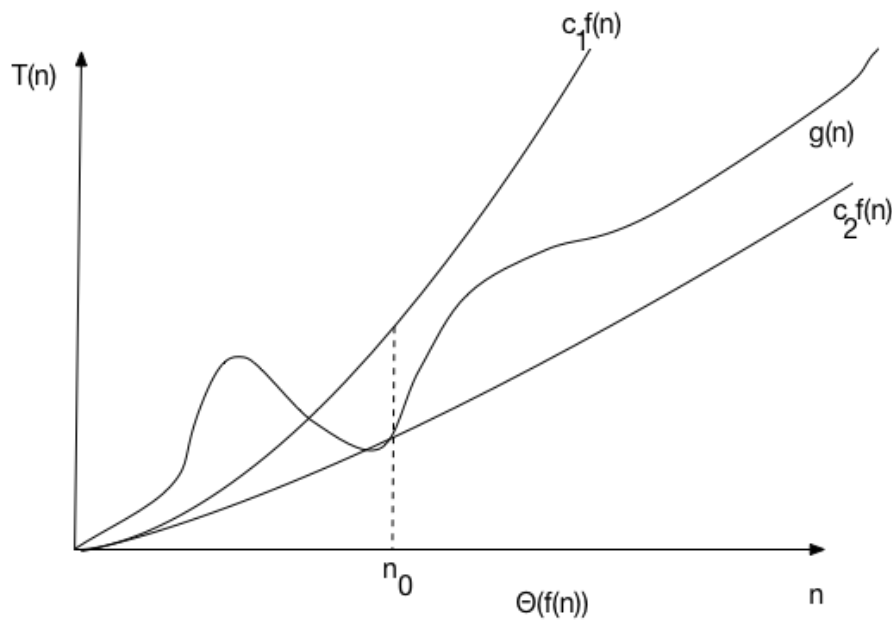
Definiamo $g(n) = \Omega(f(n))$ se e solo se $\exists c, n_0 : g(n) \geq cf(n) \forall n > n_0$



Notazione asintotica per la complessità: limitazione

$$\Theta(f(n))$$

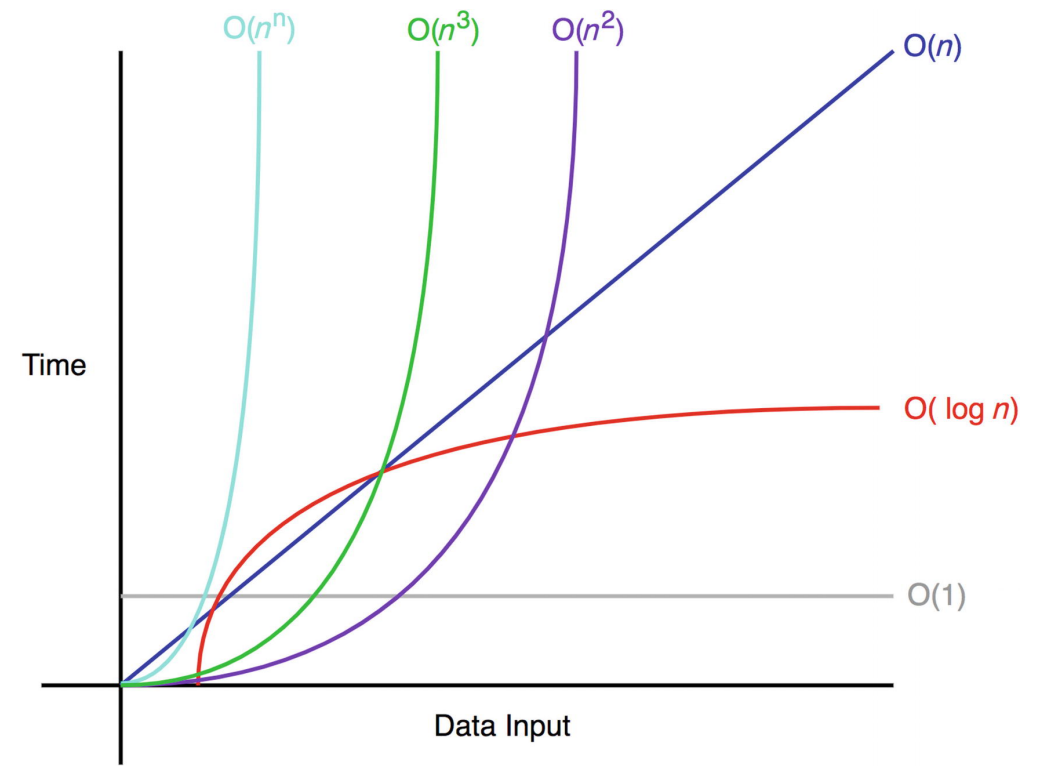
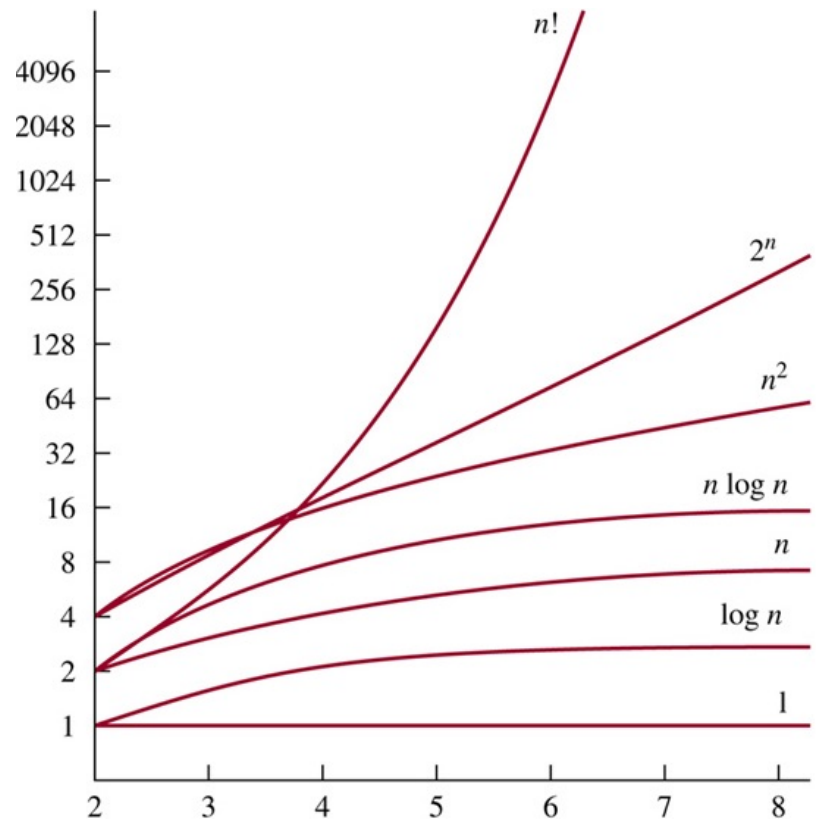
Definiamo $g(n) = \Theta(f(n))$ se e solo se $\exists c_1, c_2, n_0 : c_1 f(n) \geq g(n) \geq c_2 f(n) \forall n > n_0$



Equivale a $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$

Ordini di complessità

$f(n)$	10^1	10^2	10^3	10^4	Tipo
$\log n$	3	6	9	13	logaritmico
\sqrt{n}	3	10	31	100	sublineare
n	10	100	1000	10000	lineare
$n \log n$	30	664	9965	132877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^{12}	cubico
2^n	1024	10^{30}	10^{300}	10^{3000}	esponenziale



Esempi di ordini di complessità

- $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$ perché esistono le costanti $c = 5$ e $n_0 = 4$ per cui $g(n) \leq 5n^2 \forall n > 4$.

| $g(n)$ è anche $O(n^k)$ per $k \geq 2$, tuttavia si richiede di solito l'ordine più piccolo.

- $g(n) = 4n^2 + 4n - 1$ è $\Omega(n^2)$ perché esistono le costanti $c = 1$ e $n_0 = 1$ per cui $g(n) \geq n^2 \forall n > 1$.

| $g(n)$ è anche $\Omega(n^k)$ per $k \leq 2$, tuttavia si richiede di solito l'ordine più grande.

Proprietà della notazione asintotica

Per stimare gli ordini di grandezza non è necessario applicare le definizioni ma si possono usare le seguenti regole che semplificano i calcoli:

- **Riflessività:** per ogni costante c , $c \cdot f(n)$ è $O(f(n))$ (lo stesso per Ω e Θ)
- **Transitività:** se $g(n) = O(f(n))$ e $f(n) = O(h(n))$ allora $g(n) = O(h(n))$ (lo stesso per Ω e Θ)
- **Simmetria:** $g(n) = \Theta(f(n))$ se e solo se $f(n) = \Theta(g(n))$
- **Simmetria trasposta:** $g(n) = O(f(n))$ se e solo se $f(n) = \Omega(g(n))$
- **Somma:** $f(n) + g(n) = O(\max\{f(n), g(n)\})$ (lo stesso per Ω e Θ)
- **Prodotto:** $g(n) = O(f(n))$, $h(n) = O(q(n))$ allora $g(n) \cdot h(n) = O(f(n) \cdot q(n))$ (lo stesso per Ω e Θ)

Esempio di calcolo con le proprietà della notazione asintotica

$g(n) = 4n^2 + 4n - 1$ è $O(n^2)$ perché:

- $4n$ è $O(n)$ per la regola di riflessività
- $4n^2 = 4n \cdot n$ è $O(n \cdot n) = O(n^2)$ per la regola di prodotto
- $4n^2 + 4n - 1$ è $O(\max\{n^2, n, 1\})$ per la regola di somma

Analisi strutturale

Analisi strutturale della complessità degli algoritmi

Per gli algoritmi descritti in un linguaggio imperativo (come il C) è possibile definire delle regole che consentono di stimare la loro complessità computazionale direttamente dalla struttura del programma stesso

In particolare la stima strutturale al caso pessimo risulta essere particolarmente agevole grazie all'assorbimento degli ordini di crescita inferiori nella notazione asintotica

Guida per il calcolo del costo al caso pessimo

- La complessità di una sequenza di istruzioni è data dalla somma delle complessità delle singole istruzioni della sequenza
- Il costo di una chiamata a funzione è il costo del suo corpo più il passaggio dei parametri (qualora siano a loro volta delle operazioni non elementari)
 - le funzioni ricorsive saranno trattate a parte

Guida per il calcolo del costo al caso pessimo

- La complessità di una condizione `if (guardia) blocco1 else blocco 2` è data da $\text{costo}(\text{guardia}) + \max\{\text{costo}(\text{blocco1}), \text{costo}(\text{blocco2})\}$
- La complessità di un ciclo con un numero determinato di iterazioni `for (i = 0; i < m; i = i + 1) corpo` è data da $\sum_{i=0}^{m-1} \text{costo}(\text{corpo})_i$ costo del corpo all'iterazione i , più due volte un costo costante dovuto dalla valutazione di inizializzazione / incremento e condizione
- La complessità di un ciclo con un numero indeterminato di iterazioni `while (guardia) corpo` è data da $\sum_{i=0}^{m-1} \text{costo}(\text{guardia})_i + \text{costo}(\text{corpo})_i + \text{costo}(\text{guardia})_m$ dove m è il massimo numero delle volte in cui la guardia risulta soddisfatta

Esempio di analisi strutturale

Costo computazionale nel calcolo del minimo di un vettore (versione naïve: *don't try this at home*)

```
int min_naive(int a[], int n) {           // Costo, Num. ripetizioni
    int i, j;                             // 0      1
    bool is_min;                           // 0      1
    for (i = 0; i < n; i++) {              // 2c1     n
        is_min = true;                    // c2      n
        for (j = 0; j < n ; j++)           // 2c3     n * n
            if (a[i] > a[j])               // c4      n * n
                is_min = false;            // c5      n * n
        if (is_min)                        // c6      n
            return a[i];                   // c7      1
    }
}
```

$$T(n) = 2c_1 \cdot n + c_2 \cdot n + 2c_3 \cdot n \cdot n + c_4 \cdot n \cdot n + c_5 \cdot n \cdot n + c_6 \cdot n + c_7 = n^2(2c_3 + c_4 + c_5) + n(2c_1 + c_2 + c_6) + 1(c_7)$$

Esempio di analisi strutturale

$$\begin{aligned} T(n) &= n^2 \underbrace{(2c_3 + c_4 + c_5)}_{c'} + n \underbrace{(2c_1 + c_2 + c_6)}_{c''} + 1 \underbrace{(c_7)}_{c'''} \\ &= O(n^2) + O(n) + O(1) = O(n^2) \end{aligned}$$

(per le regole di **riflessività** e **somma** della notazione asintotica)

Costo computazionale: caso medio

Supponendo che i valori nell'array siano distribuiti uniformemente, il minimo ha la probabilità $\frac{1}{n}$ di trovarsi in un punto j del vettore e il costo dell'algoritmo, per la posizione j è dato da:

$$jn\left(1 - \frac{1}{n}\right)^{j-1}$$

- $j \cdot n$ è il numero di elementi verificati (il numero di iterazioni del ciclo esterno j moltiplicato per quelle del ciclo interno n)
- $\left(1 - \frac{1}{n}\right)^{j-1}$ è la probabilità che il minimo non fosse nessun elemento in posizione minore di j (la probabilità che non sia il primo è $\left(1 - \frac{1}{n}\right)$, che non sia il secondo $\left(1 - \frac{1}{n}\right) \cdot \left(1 - \frac{1}{n}\right)$, che non sia il terzo $\left(1 - \frac{1}{n}\right)^3$, e così via)

Costo computazionale: caso medio

Il **valore atteso** (la media) del numero di valori verificati è dato dalla somma di tali valori per tutte le posizioni j , diviso per il numero n di valori

$$\mathbb{E}[T(n)] = \sum_{j=1}^n \frac{jn \left(1 - \frac{1}{n}\right)^{j-1}}{n} = \left(1 - 2\left(1 - \frac{1}{n}\right)^n\right)n^2$$

si ha

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e}$$

dunque

$$\mathbb{E}[T(n)] \approx \left(1 - \frac{2}{e}\right)n^2 = O(n^2)$$

Esempio di analisi strutturale

Dato un vettore a di dimensione n trovarne il valore minimo (algoritmo più efficiente)

```
int minimo(int a[], int n) {           // Costo, Num. volte
    int i, min = a[0];                 // c1          1
    for (i = 1; i < n; i++) {          // 2c2         n - 1
        if (a[i] < min)                // c3          n - 1
            min = a[i];                // c4          n - 1
    }
    return min;                        // c5          1
}
```

$$\begin{aligned} T(n) &= c_1 + 2c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_5 \\ &= (n - 1) \underbrace{(2c_2 + c_3 + c_4)}_{c'} + 1 \underbrace{(c_1 + c_5)}_{c''} = O(n) \end{aligned}$$

Nel caso medio: risparmio j assegnamenti `min = a[i]`, ma asintoticamente il comportamento è sempre $O(n)$

Costo computazionale nella ricerca di un elemento nel vettore

Dato un vettore a di dimensione n , verificare se l'elemento v è presente nel vettore.

```
bool cerca(int a[], int n, int v) { // Costo, Num. volte
    int i;                          // 0      1
    for (i = 0; i < n; i++)          // 2c1    n
        if (a[i] == v)              // c2     n
            return true;             // c3    max 1
    return false;                    // c4    max 1
}
```

$$\begin{aligned} T(n) &= 2c_1 \cdot n + c_2 \cdot n + c_3 + c_4 \\ &= \underbrace{n(2c_1 + c_2)}_{c'} + 1 \cdot \underbrace{\max\{c_3, c_4\}}_{c''} = O(n) \end{aligned}$$

Costo computazionale nel calcolo della ricerca binaria

Dato un vettore a di dimensione n con elementi **ordinati**, verificare se l'elemento v è presente nel vettore

```
bool ricerca_binaria(int a[], int n, int v, int i, int j) { // Costo,   Num (i > j), Num (i <= j)
    int m; // 0,   1,   1
    if (i > j) // c1,   1,   1
        return false; // c2,   1,   0
    else {
        m = floor((i + j) / 2); // c3,   0,   1
        if (a[m] == v) // c4,   0,   1
            return true; // c5,   0,   max 1
        else if (a[m] < v) // c6,   0,   1
            return ricerca_binaria(a, n / 2, v, m + 1, j); // c7 + T(n/2) 0,   max 1
        else
            return ricerca_binaria(a, n / 2, v, i, m - 1); // c8 + T(n/2) 0,   max 1
    }
}
```

Costo computazionale nel calcolo della ricerca binaria

Assunzioni (**caso pessimo**): l'elemento cercato non è presente

$$T(n) = \begin{cases} c_1 + c_2 = c' & \text{se } i > j \text{ (cioé } n = 0) \\ T(n/2) + \underbrace{1(c_3 + c_4 + c_5 + c_6 + \max\{c_7, c_8\})}_{c''} & \text{se } i > j \text{ (cioé } n > 0) \end{cases}$$

Risolviamo la relazione di ricorrenza tramite espansione: applichiamo la definizione ricorsiva cercando di scoprire qualche regolarità:

$$\begin{aligned} T(n) &= \underbrace{T(n/2)}_{T(n/4)+c''} + c'' = \underbrace{T(n/4)}_{T(n/8)+c''} + 2c'' \\ &= \underbrace{T(n/8)}_{T(n/16)+c''} + 3c'' = \dots = \underbrace{T(n/2^k)}_{\text{al passo } k} + kc'' \end{aligned}$$

Costo computazionale nel calcolo della ricerca binaria

Il caso di base si raggiunge quando $n/2^k$ è pari all'argomento di base della ricorsione, ovvero $n/2^k = 0$ (divisione intera) cioè quando $k = \log_2 n + 1$, per tale valore

$$\begin{aligned} T(n) &= T(0) + kc'' = c' + kc'' \\ &= c' + (\log_2 n + 1)c'' = \log_2 nc'' + 1(c' + c'') = O(\log n) \end{aligned}$$

Costo computazionale nel calcolo dei numeri di Fibonacci con funzione ricorsiva

```
int fibonacci(int n) {                                // Costo, Num. volte
    if (n == 0 || n == 1)                             // c1      1
        return 1;                                     // c2      1
    else
        return fibonacci(n - 1) + fibonacci(n - 2); // c3      T(n - 1) + T(n - 2)
}
```

Abbiamo un'equazione ricorsiva:

$$T(n) = \begin{cases} T(n-1) + T(n-2) + c' & \text{se } n > 2 \\ c'' & \text{se } n \leq 2 \end{cases}$$

Costo computazionale nel calcolo dei numeri di Fibonacci con funzione ricorsiva

Una tecnica per risolvere l'equazione è utilizzare una maggiorazione e un'espansione, dato che $T(n)$ è sicuramente una funzione non decrescente (perché?)

$$\begin{aligned} T(n) &= T(n-1) + \overbrace{T(n-2)}^{\leq T(n-1)} + c' \\ &\leq 2 \overbrace{T(n-1)}^{T(n-2)+T(n-3)+c'} + c' = 2T(n-2) + \overbrace{2T(n-3)}^{\leq T(n-2)} + 2c' + c' \\ &\leq 4T(n-2) + 2c' + c' = 4T(n-3) + 4T(n-4) + 4c' + 2c' + c' \\ &\leq 8T(n-3) + 4c' + 2c' + c' = \dots \\ &\vdots \\ &\leq 2^k T(n-k) + c' \sum_{i=0}^{k-1} 2^i \end{aligned}$$

Costo computazionale nel calcolo dei numeri di Fibonacci con funzione ricorsiva

Quindi, poiché l'espansione termina al caso base dell'argomento di $T(n)$, $n = 2$, ovvero quando $n - k \leq 2$, ossia quando almeno $k = n - 2$, pertanto

$$\begin{aligned} T(n) &\leq 2^{n-2}T(1) + c' \sum_{i=0}^{n-3} 2^i = 2^{n-2}c' + c' \sum_{i=0}^{n-3} 2^i \\ &\leq 2^{n-2}c' + c'(2^{n-2} - 1) \\ &\leq 2^{n-2}2c' - c' = 2^{n-1}c' - c' = O(2^n) \end{aligned}$$

Vale la proprietà:

$$\sum_{i=0}^{n-1} q^i = \frac{q^n - 1}{q - 1}$$

Costo computazionale nel calcolo dei numeri di Fibonacci con funzione ricorsiva

In realtà la maggiorazione potrebbe non essere sufficientemente stretta, ma utilizzando la maggiorazione opposta per cercare un limite inferiore.

$$\begin{aligned} T(n) &= \overbrace{T(n-1)}^{\geq T(n-2)} + T(n-2) + c' \\ &\geq 2 \overbrace{T(n-2)}^{T(n-3)+T(n-4)+c'} + c' = 2 \overbrace{T(n-3)}^{\geq T(n-4)} + 2T(n-4) + 2c' + c' \\ &\geq 4T(n-4) + 2c' + c' = 4T(n-5) + 4T(n-6) + 4c' + 2c' + c' \\ &\geq 8T(n-8) + 4c' + 2c' + c' = \dots \\ &\vdots \\ &\geq 2^k T(n-2k) + c' \sum_{i=0}^{k-1} 2^i \end{aligned}$$

Costo computazionale nel calcolo dei numeri di Fibonacci con funzione ricorsiva

Il caso base in cui l'espansione termina è ora quando $n - 2k \leq 2$, ovvero quando almeno $k = \frac{n}{2} - 1$ e

$$\begin{aligned} T(n) &\geq 2^{\frac{n}{2}-1} T(2) + c' \sum_{i=0}^{\frac{n}{2}-2} 2^i = 2^{\frac{n}{2}-2} c' + c' \sum_{i=0}^{\frac{n}{2}-2} 2^i \\ &\geq 2^{\frac{n}{2}-2} c' + c' (2^{\frac{n}{2}-2} - 1) \\ &\geq 2^{\frac{n}{2}-2} 2c' - c' = 2^{\frac{n}{2}-1} c' - c' = \Omega(1.41592^n) = O(2^n) \end{aligned}$$

Pertanto $T(n) = \Theta(2^n)$