POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

Algoritmi e Problem-solving

Paolo Camurati

Algoritmo

Sequenza finita di istruzioni che:

- risolvono un problema
- soddisfano i seguenti criteri:
  - ricevono valori in ingresso
  - producono valori in uscita
  - sono chiare, non ambigue ed eseguibili
  - terminano dopo un numero finito di passi
- operano su strutture dati

Algoritmo: da al-Khwarizmi, matematico persiano/uzbeko del IX secolo d.C.

Muḥammad ibn Mūsā al-Khwārizmī
محمد بن موسى الخوارزمي

Algoritmo vs. procedura

Se, per ogni possibile valore di ingresso, la terminazione è garantita in un numero finito di passi, allora

⬇

algoritmo

altrimenti

⬇

procedura

La congettura di Collatz (1937)

Data la funzione f(n) così definita:

$$f(n) = \begin{cases} n/2 \text{ per n pari} \\ 3n + 1 \text{ per n dispari} \end{cases}$$

dato un qualsiasi numero naturale n, la funzione f(n) convergerà ad 1 in un numero finito di passi?

Non siamo in grado di rispondere!

n = 11

11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

converge in 15 passi

n = 27

Converge in 111 passi

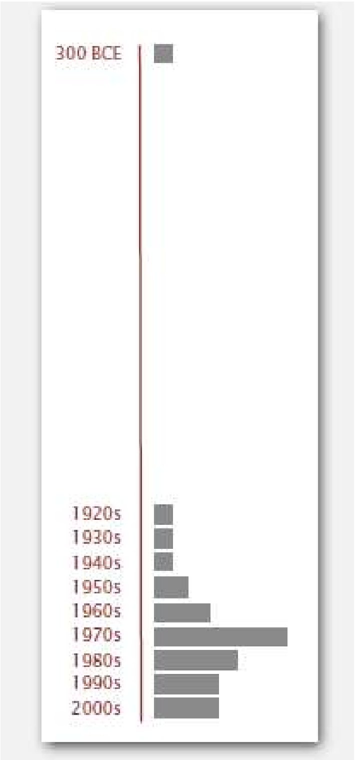
Non si può garantire che con n qualsiasi converga. Non abbiamo alcuna garanzia che termini, quindi procedura e non algoritmo.

```c
#include <stdio.h>
int main() {
  int n;
  printf("Input natural number:  ");
  scanf("%d", &n);

  while (n > 1) {
    printf("%d   ", n);
    if ((n%2) ==1)
      n = 3*n + 1;
    else
      n = n/2;
  }
  printf("%d \n", n);
  return 0;
}
```

Gli algoritmi nella storia

- Moltiplicazioni egizie (papiri di Rhind e Ahmes, circa 1900 a.C.)
- Massimo Comun Divisore (algoritmo di Euclide, IV secolo a.C.)
- formalizzazione di Church e Turing (XX secolo, anni ‘30)
- sviluppi recenti



Moltiplicazioni egizie

Assunzioni:

- non servono le tabelline pitagoriche
- basta sapere moltiplicare per 2, quindi anche conoscere le potenze di 2.

Dati due numeri naturali *x* e *y*, si costruiscono 2 colonne: a sinistra quella dei multipli di x per potenze di 2, a destra quella delle potenze di $2 \leq y$.

Determinare le potenze di 2 che sommate danno y, il risultato è la somma delle righe corrispondenti della colonna di sinistra.

Esempio: x = 18 y = 33

$18 \cdot 33 = 18 + 576 = 594$

Matematicamente: $33 = 32 + 1 = 2^5 + 2^0$

$18 \cdot 33 = 18 \cdot (1 + 32) = 18 + 576 = 594$

18	1
36	2
72	4
144	8
288	16
576	32

Il Massimo Comun Divisore

Il massimo comun divisore *gcd* di due interi *x* e *y* non entrambi nulli è il più grande dei divisori comuni di *x* e *y*. Si assuma che inizialmente *x > y.*
Algoritmo inefficiente basato sulla scomposizione in fattori primi:

x = $p_1^{e_1} \cdot p_2^{e_2} \cdot \cdot \cdot p_r^{e_r}$ y = $p_1^{f_1} \cdot p_2^{f_2} \cdot \cdot \cdot p_s^{f_s}$

gcd(x,y) = $\prod p_i^{\min(e_i, f_i)}$

Esempio: gcd(96,54) = 6

96 = $2^5 \cdot 3^1$
54 = $2^1 \cdot 3^3$
gcd(96, 54) = $2^1 \cdot 3^1$ = 6

Algoritmo di Euclide

Ricorsivo! Argomento di Algoritmi al II anno

Versione 1: sottrazione

- se x > y
  - gcd(x, y) = gcd(x-y, y)
- altrimenti
  - gcd(x, y) = gcd(x, y-x)
- terminazione:
  - se x=y ritorna x

gcd(96, 54) = gcd(42, 54)
gcd(42, 54) = gcd(42, 12)
gcd(42, 12) = gcd(30, 12)
gcd(30, 12) = gcd(18, 12)
gcd(18, 12) = gcd(6, 12)
gcd(6, 12) = gcd(6, 6) = 6

Versione 2 Euclide-Lamé (1844)-Dijkstra: resto della divisione intera (%)

se x > y

gcd(x, y) = gcd(y, x%y)

terminazione:

se y = 0 ritorna x

gcd(96, 54) = gcd(54, 42)
gcd(54, 42) = gcd(42, 12)
gcd(42, 12) = gcd(12, 6)
gcd(12, 6) = gcd(6, 0) = 6

Perché gli algoritmi?

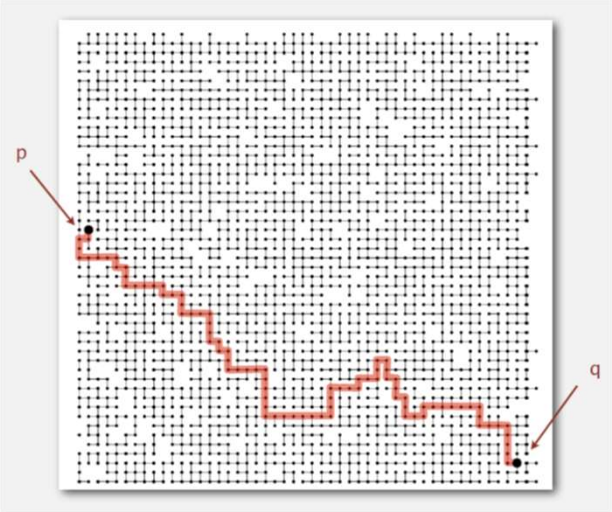
- per risolvere problemi in moltissimi campi
- per fare qualcosa di altrimenti impossibile
- per curiosità intellettuale
- per programmare bene
- per creare modelli
- per divertimento o per denaro.

Per risolvere problemi in moltissimi campi:

- Internet: Web search, packet routing, distributed file sharing
- Biologia: genoma umano
- Computer: strumenti CAD, file systems, compilatori
- Grafica: realtà virtuale, videografica
- Multimedia: MP3, JPG, DivX, HDTV
- Social Networks: recommendations, news feed, pubblicità
- Sicurezza: e-commerce, cellulari
- Fisica: simulazione di collisione di particelle …

Per fare qualcosa di altrimenti impossibile a mano:

- in questa rete, i 2 punti evidenziati sono connessi tra di loro (network connectivity)?



Per programmare bene:

“ I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. ”

— Linus Torvalds (creator of Linux)

Per creare modelli:

- in molte scienze i modelli computazionali stanno sostituendo quelli matematici

$$E = mc^2$$

$$F = ma \qquad F = \frac{G m_1 m_2}{r^2}$$

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \Psi(r) = E \, \Psi(r)$$

```
for (double t = 0.0; true; t = t + dt)
   for (int i = 0; i < N; i++)
   {
      bodies[i].resetForce();
      for (int j = 0; j < N; j++)
         if (i != j)
            bodies[i].addForce(bodies[j]);
   }
```

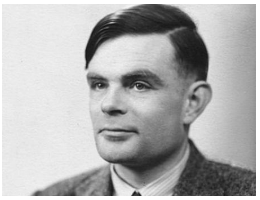
Formule matematiche   Modello computazionale

Il modello computazionale

- Chi o cosa esegue un algoritmo?
  - uomo
  - macchina
- Ci sono limiti sulla potenza delle macchine che possiamo costruire?
- Esiste un modello universale di computazione?

La macchina di Turing

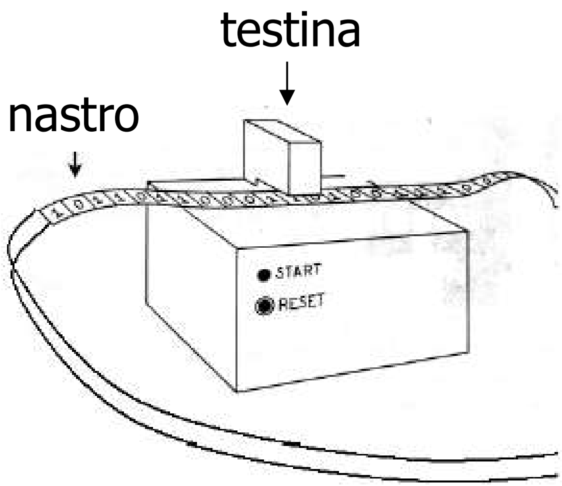
La macchina di Turing



La Macchina di Turing è un modello che, data una funzione f e un *input*, al termine della computazione genera il corrispondente *output*.

È definita da:

- un nastro
- una testina
- uno stato interno
- un programma
- uno stato iniziale

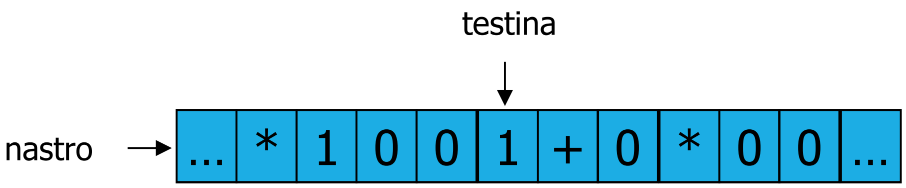


Il nastro:

- memorizza input, output e risultati intermedi
- ha lunghezza infinita ed è diviso in celle
- ogni cella contiene un simbolo $\in$ alfabeto

La testina:

- punta a una cella del nastro
- legge o scrive la cella corrente
- si sposta di 1 cella a DX o SX



Lo stato interno è la configurazione della macchina in funzione dell'input corrente e della «storia» passata.
La macchina, in funzione dello stato e dell'input correnti, scrive un valore sul nastro e sposta la testina a DX o SX (programma).

La tesi di Church-Turing (1936)

«La Macchina di Turing può calcolare qualsiasi funzione che possa essere calcolata da una macchina fisicamente realizzabile.»

Tesi e non teorema perché è un’affermazione sul mondo fisico non soggetta a prova, però in 80 anni non si sono trovati controesempi.

Tutti i modelli computazionali finora trovati sono equivalenti alla Macchina di Turing.

Il problem-solving

Attività del pensiero che:

- un organismo o un dispositivo di intelligenza artificiale mettono in atto
- per raggiungere una condizione desiderata a partire da una condizione data
- altamente creativa, di natura progettuale.

Approccio al problem-solving

1. analisi del problema:
   lettura delle specifiche, comprensione del problema, identificazione della classe di problemi noti cui il problema in esame appartiene
2. identificazione delle metodologie:
   scelta tra paradigmi algoritmici noti (incrementale, divide et impera, programmazione dinamica, greedy, etc.)
3. selezione di un approccio:
   scelta dell'approccio migliore in base ad un'analisi di complessità

4. decomposizione in sottoproblemi:
   identificazione dei sottoproblemi e delle loro interazioni in un’ottica di modularità
5. definizione dell'algoritmo risolutivo:
   identificazione della sequenza di passi elementari e dei dati su cui opera e dimostrazione della correttezza
6. riflessione critica:
   ripensamento per identificare criticità, possibili migliorie, etc.

Problemi computazionali

- Modelli formali cui è associato un insieme di domande a cui dare risposta mediante le elaborazioni di un computer che esegue un programma
- insieme di istanze infinite del problema, a ciascuna delle quali è associata una soluzione.
- istanza: problema su dati specifici.

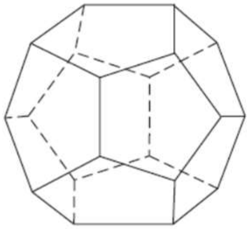
Tipologie di problemi

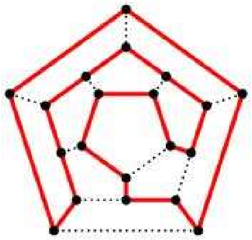
Problemi di decisione: problemi che ammettono una risposta sì/no

- dati 2 interi x e y, x è un divisore di y?
- dato un intero positivo x>1, x è primo?
- dato un intero positivo n, esistono 2 interi positivi e >1 p e q tali che n = pq?

Problemi di ricerca: esiste e quale è una soluzione valida?
La soluzione si trova in uno spazio delle soluzioni eventualmente infinito:

- gioco di Hamilton (1859): in un dodecaedro (per alcuni un icosaedro), assegnando ad ogni vertice un nome di città, trovare un percorso che tocchi tutte le città una e una sola volta e ritorni a quella di partenza
- Teoria dei Grafi: ciclo Hamiltoniano: dato un grafo non orientato, esiste e quale è un ciclo semplice che contiene tutti i vertici?
- quale è il k-esimo numero primo?
- dato un vettore di interi, riordinarlo in ordine ascendente.

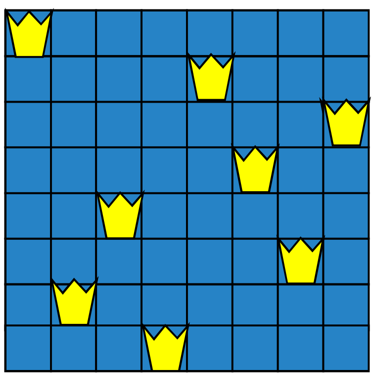




Problemi di verifica: data una soluzione (certificato), appurare che è davvero tale:

- le 8 regine sono disposte in modo che nessuna metta sotto scacco le altre?
- la griglia di destra soddisfa le regole del Sudoku (cifre da 1 a 9 su righe, colonne e quadrati 3x3 senza ripetizioni)

8 regine



Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Problemi di ottimizzazione: se esiste, quale è la soluzione migliore?

- cammini minimi: dato un grafo orientato e pesato, quale è il cammino semplice di lunghezza minima, se esiste, tra i nodi i e j?
- Commesso viaggiatore (Travelling Salesman Problem): ciclo hamiltoniano di lunghezza minima.

I problemi di ottimizzazione possono avere una versione di decisione introducendo un limite sulle soluzioni valide:

- cammini minimi: dato un grafo orientato e pesato, dati i nodi i e j e una distanza massima d, esiste un cammino semplice tra i e j di lunghezza ≤ d?

Ogni problema di ottimizzazione è almeno altrettanto difficile della sua versione di decisione.

I problemi di decisione

Possono essere:

- decidibili (esiste un algoritmo che li risolve)
  - determinare se un numero è primo

Algoritmo 1: si provano i numeri tra 2 e al massimo n, si termina:

- o perché si trova un fattore (n%fact==0, n non è primo)
- o perché fact diventa n (n è primo)

```
int Prime(int n) {
  int fact;
  if (n == 1)
    return 0;
  fact = 2;
  while (n % fact != 0)
    fact  = fact +1;
  return (fact == n);
}
```

Algoritmo 2: si provano i fattori tra 2 e √n, si termina:

- o non appena si trova un fattore (n%fact==0, n non è primo)
- o quando si è esaurito il ciclo (n è primo)

```
int PrimeOpt(int n) {
  int fact, found=0;
  if (n == 1)
    return 0;
  fact = 2;
  while (fact <= sqrt(n) && found ==0) {
    if (n % fact == 0)
      found = 1;
     else
      fact  = fact +1;
  }
  return found;
}
```

In cosa differiscono i 2 algoritmi? Nel numero di passi massimo che potranno eseguire:

- Algoritmo 1: al massimo n passi
- Algoritmo 2: al massimo $\lceil \sqrt{n} \rceil$ passi

$\Rightarrow$ differiscono nella **COMPLESSITÀ**!

I problemi di decisione

Possono essere:

- indecidibili (non esiste alcun algoritmo che li risolve)
  - dato un algoritmo A e un dato D, entrambi arbitrari, stabilire se la computazione A(D) termina in un numero finito di passi (Turing halting problem, 1937)
  - Congettura di Goldbach (XVII secolo): ogni numero intero pari maggiore di 2 è la somma di due numeri primi p e q

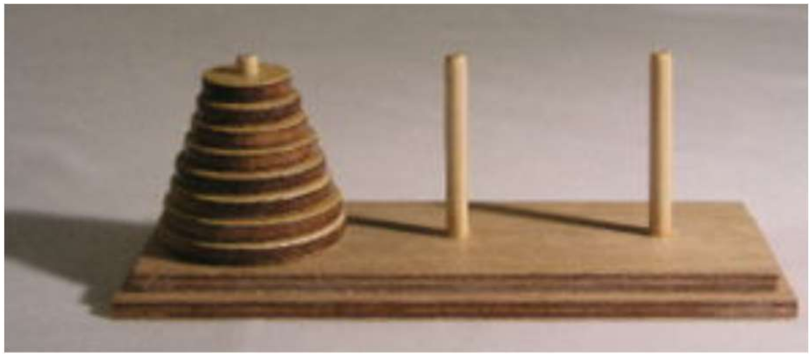
$$\forall\ n \in \aleph, (n > 2) \wedge (n \text{ pari}) \Rightarrow (\exists\ p, q \in \wp \in, n = p + q)$$

```
#define  upper 20
```

```
void Goldbach(void) {
  int n = 2, counterexample, p, q;
  do {
    n = n + 2;
    printf("I try for n = %d\n", n);
    counterexample = 1;
    for (p = 2; p <= n-2; p++){
      q = n - p;
      if (Prime(p) == 1 && Prime(q) == 1){
        counterexample = 0;
        printf("%d %d\n", p, q);
      }
    }
  } while (counterexample == 0 && n < upper);
    if (counterexample == 1)
      printf("Counterexample is: %d \n", n);
    else
      printf("Until n= %d none found\n", upper);
  return;
}
```

I problemi di decisione decidibili possono essere:

- trattabili, cioè risolvibili in tempi “ragionevoli”:
  - ordinare un vettore di n interi
- intrattabili, cioè non risolvibili in tempi “ragionevoli”:
  - le Torri di Hanoi



Le Torri di Hanoi (E. Lucas 1883)

- Configurazione iniziale:
  - vi sono 3 pioli, 3 dischi di diametro decrescente sul primo piolo
- Configurazione finale:
  - 3 dischi sul terzo piolo
- Regole:
  - accesso solo al disco in cima
  - sopra ogni disco solo dischi più piccoli
- Generalizzabile a n dischi e k pioli.

La Classe P

Problemi di decisione decidibili e trattabili

$\Leftrightarrow$

$\exists$ un algoritmo polinomiale che li risolve (Tesi di Edmonds-Cook-Karp, anni '70)

Polinoniale: algoritmo che, operando su n dati, data una costante $c > 0$, termina in un numero di passi limitato superiormente da n^c

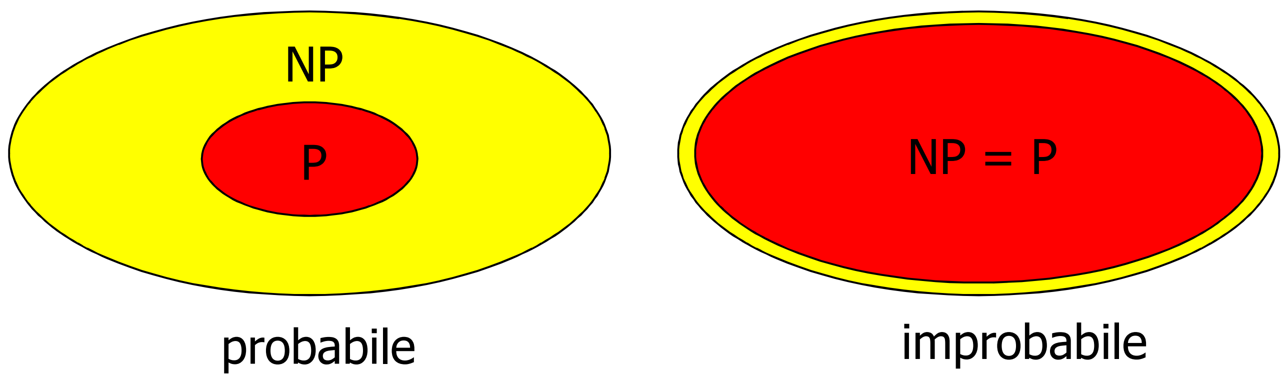
- in pratica c non deve eccedere 2.

La Classe NP

- Esistono problemi di decisione decidibili per cui conosciamo algoritmi di soluzione esponenziali, ma non conosciamo algoritmi polinomiali. Non possiamo però escludere che esistano
- Conosciamo algoritmi polinomiali per **verificare** che una soluzione (*certificato*) ad un'istanza è davvero tale
  - Sudoku, soddisfacibilità di una funzione booleana

- PS: NP sta per Non-deterministico Polinomiale e fa riferimento alla Macchina di Turing non deterministica.

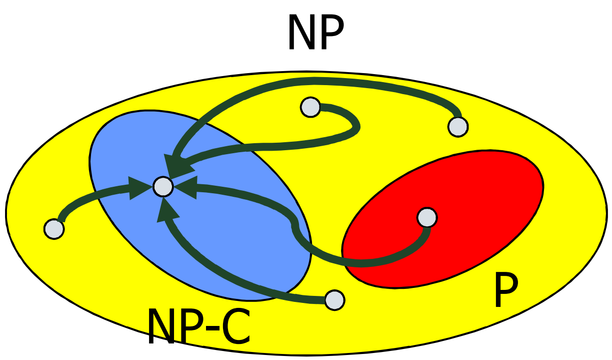
$P \subseteq NP$, ma non è noto se P è un sottoinsieme proprio di NP o se al limite coincide con esso. È probabile che sia un sottoinsieme proprio.



La Classe NP-C

Un problema è NP-**completo** se:

- è NP
- ogni altro problema in NP è riducibile ad esso attraverso una trasformazione polinomiale



Se un problema NP-completo fosse risolvibile con un algoritmo polinomiale, allora con trasformazioni polinomiali si troverebbero algoritmi polinomiali per tutti i problemi NP.

MOLTO IMPROBABILE!

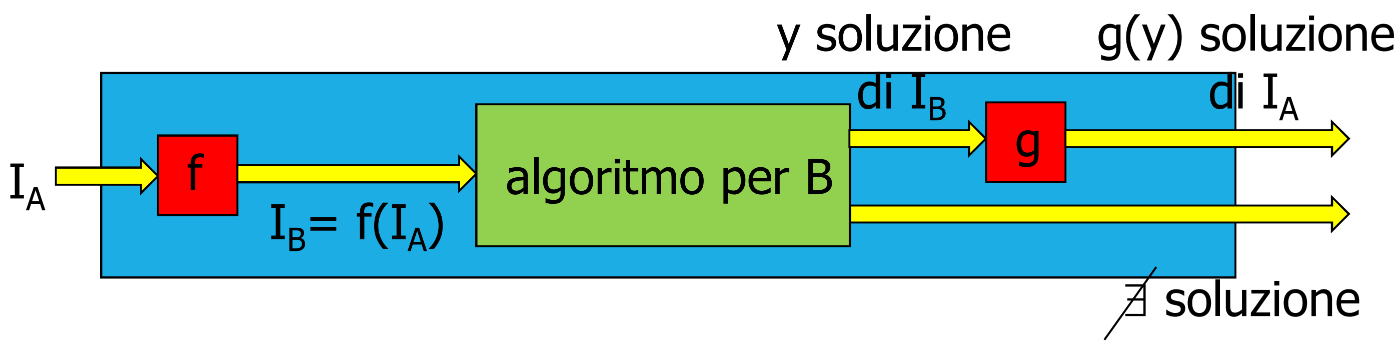
L'esistenza di NP-C rende probabile che P $\subset$ NP

Esempio di problema NP-C: soddisfacibilità

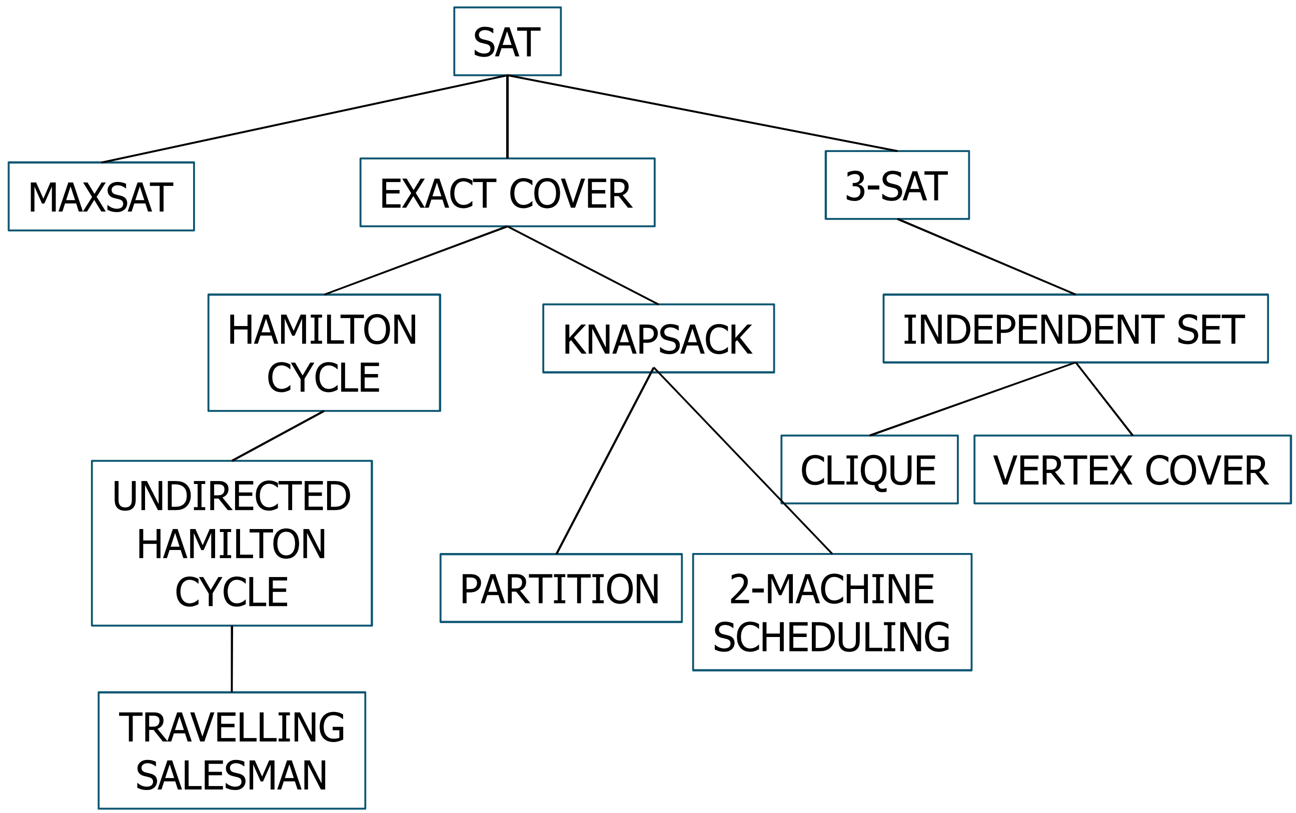
data una funzione booleana, determinare se esiste una qualche combinazione di valori delle variabili di ingresso per cui la funzione risulti vera.

Dati 2 problemi di decisione A e B, una riduzione polinomiale da A a B ($A \leq_P B$) è una procedura che trasforma ogni istanza I_A di A in una istanza I_B di B:

- con costo polinomiale
- tale che la risposta per I_A è sì se e solo se la risposta di I_B è sì



Esempi di riduzioni



Il Grafo

Definizione: $G = (V,E)$

- V: insieme finito e non vuoto di vertici (contenenti dati semplici o composti)
- E: insieme finito di archi, che definiscono una relazione binaria su V

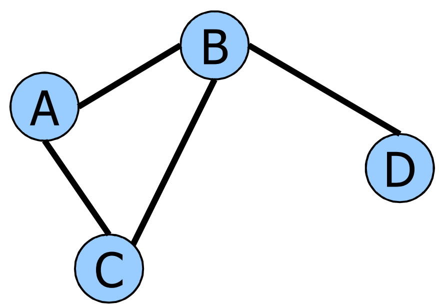
Grafi orientati/non orientati:

- orientati: arco = coppia ordinata di vertici $(u, v) \in E$ e $u, v \in V$
- non orientati: arco = coppia non ordinata di vertici $(u, v) \in E$ e $u, v \in V$

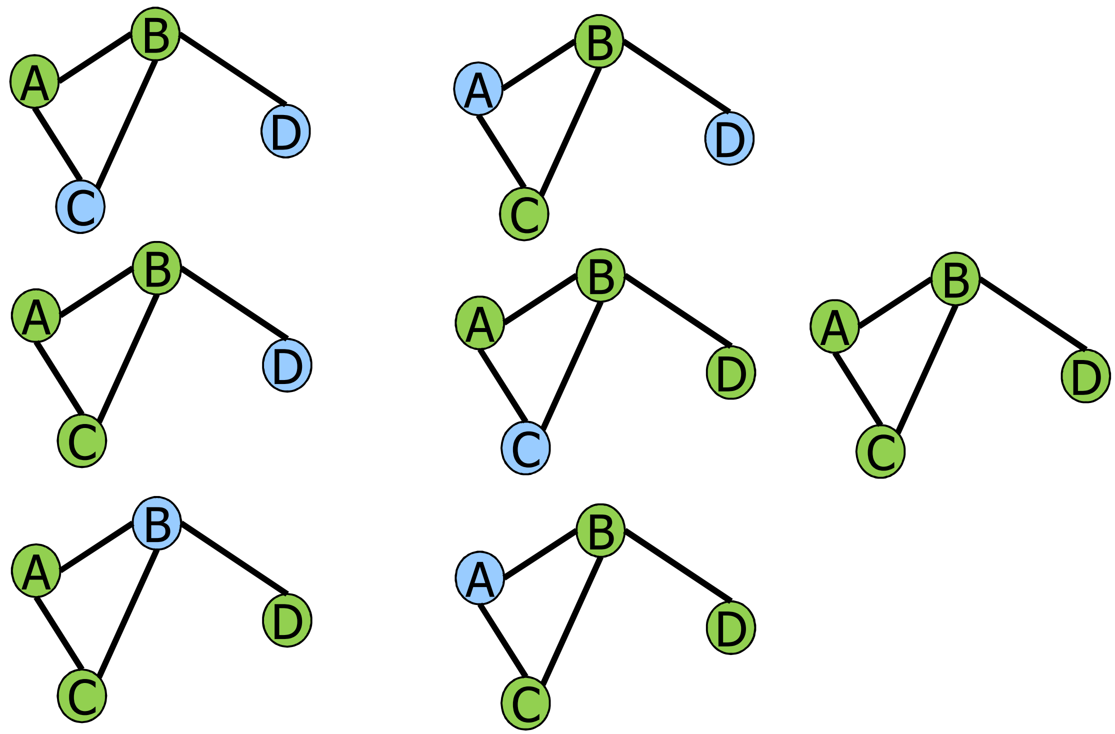
Vertex-cover

Dato un grafo non orientato G = (V, E), un vertex-cover è un sottoinsieme dei vertici $W \subseteq V$ tali che per tutti gli archi $(u,v) \in E$ o $u \in W$ o $v \in W$

Esempio

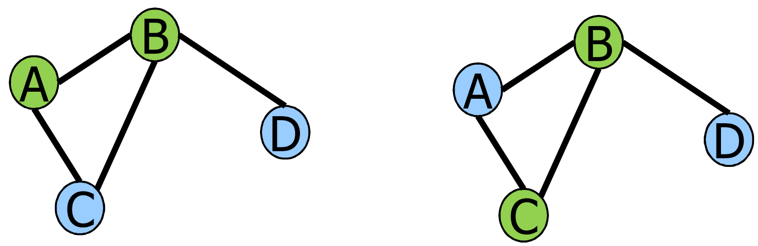


7 soluzioni



- Problema di ricerca: trovare un vertex-cover
- Problema di ottimizzazione: trovare un vertex-cover di cardinalità minima
- Problema di decisione: esiste un vertex-cover di cardinalità $\leq$ k?

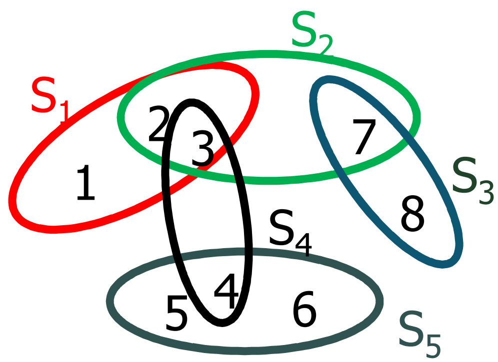
Problema di decisione con k = 2
2 soluzioni



Set-cover

Problema di decisione:
dato un insieme U di elementi, una collezione $S_1, S_2, \ldots, S_n$ di suoi sottoinsiemi e un intero k, esiste una collezione di al massimo k sottoinsiemi la cui unione sia U?

Esempio



U = {1,2,3,4,5,6,7,8}

S_1 = {1,2,3}

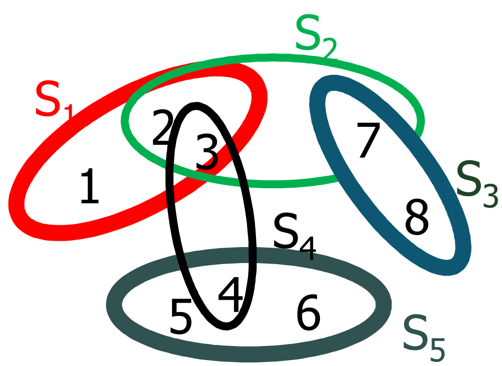
S_2 = {2,3,7}

S_3 = {7,8}

S_4 = {3,4}

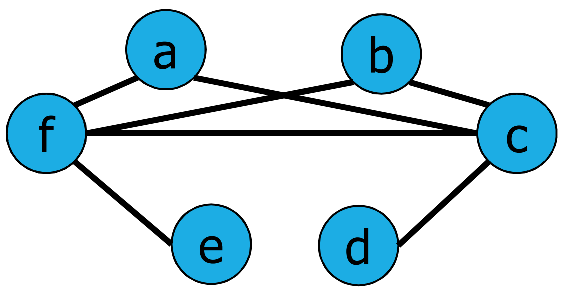
S_5 = {4,5,6}

Soluzione per k = 3



Esempio

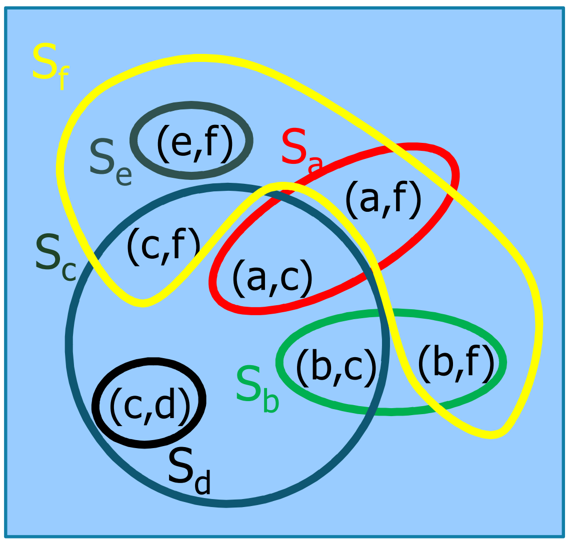
Problema di decisione: dato il grafo non orientato, esiste un vertex-cover di cardinalità ≤ 2?



$G = (V, E)$
$V = \{a,b,c,d,e,f\}$
$E = \{(a,c),(a,f),(b,c)$
$(b,f),(c,d),(c,f)(e,f)\}$

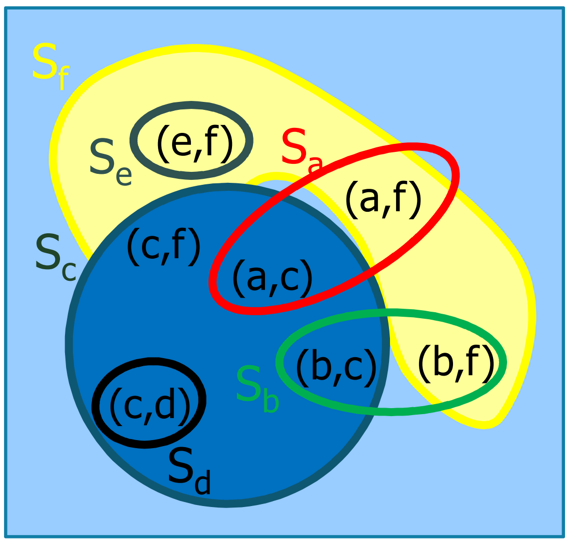
vertex-cover ≤P set-cover

Creare un problema di decisione set-cover con U = E e
S_i= {archi che insistono sul vertice i}



$U = \{(a,c),(a,f),(b,c)$
$(b,f),(c,d),(c,f)(e,f)\}$
$S_a = \{(a,c),(a,f)\}$
$S_b = \{(b,c),(b,f)\ \}$
$S_c = \{(a,c),(b,c),(c,d),(c,f)\}$
$S_d = \{(c,d)\}$
$S_e = \{(e,f)\}$
$S_f = \{(a,f),(b,f),(c,f)(e,f)\}$

Risolvere il problema di set-cover



$U = \{(a,c),(a,f),(b,c)$
$(b,f),(c,d),(c,f)(e,f)\}$

$S_a = \{(a,c),(a,f)\}$

$S_b = \{(b,c),(b,f)\ \}$

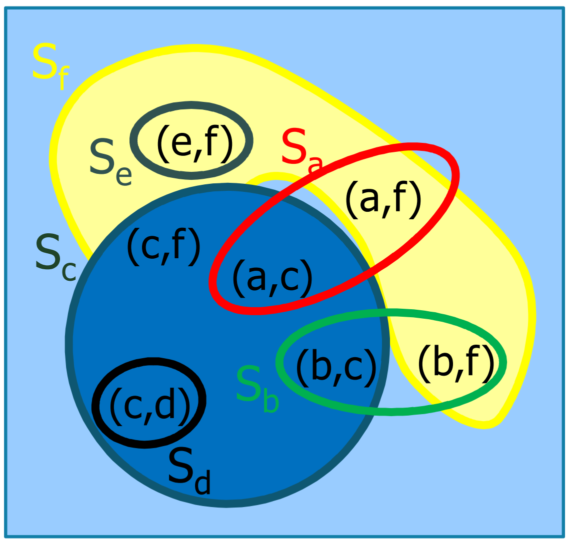
$S_c = \{(a,c),(b,c),(c,d),(c,f)\}$

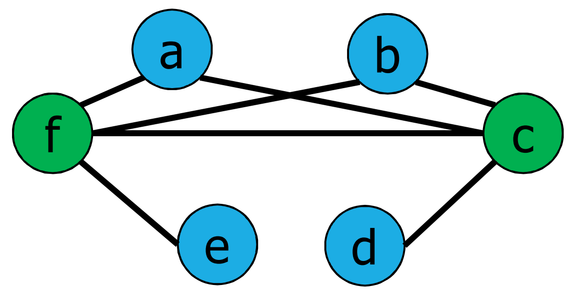
$S_d = \{(c,d)\}$

$S_e = \{(e,f)\}$

$S_f = \{(a,f),(b,f),(c,f)(e,f)\}$

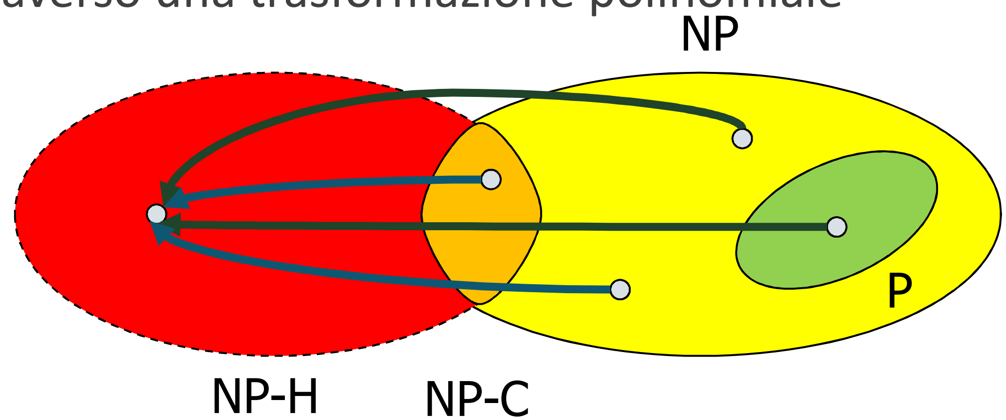
Risolvere il problema di vertex-cover

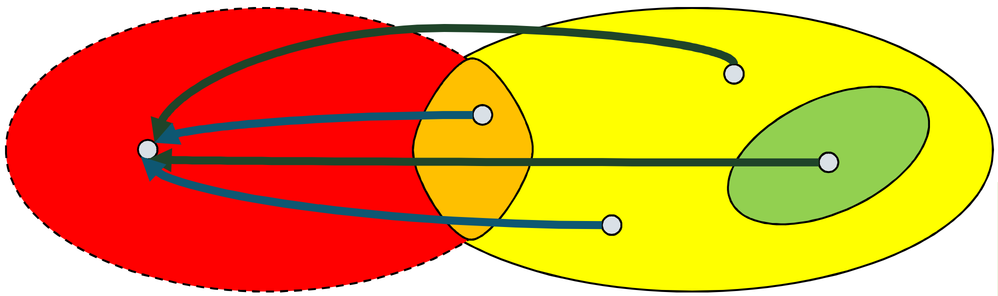




La Classe NP-H

- Un problema è NP-**hard** se ogni problema in NP è riducibile ad esso in tempo polinomiale (anche se non appartiene ad NP*)*
  - ogni altro problema in NP è riducibile ad esso attraverso una trasformazione polinomiale





NP:
- Fattorizzazione
- Isomorfismo grafi

P:
- Connettività
- grafi
- Primalità
- Determinante

NP-C:
- Soddisfacibilità
- Ciclo di Hamilton
- Clique

NP-H
- Permanente di una matrice

Il Problem-solving in questo corso

- Problem-solving **elementare**
- Problemi in classe P
- Criteri di classificazione:
  - approccio usuale: in base alle strutture dati e/o ai costrutti di controllo utilizzati
  - approccio alternativo più vicino all'utente e più lontano dal programmatore in base:
    - alla tipologia del problema
    - all'ambito applicativo: tipo di informazioni trattate e classe di problema
    - alla struttura dati utilizzata
    - alle strategie algoritmiche.

Tipologie di problemi in questo corso

- decisione:
  - problemi per cui la risposta a ogni istanza è binaria.
  - Esempio: dato un numero naturale $n > 1$, n è primo?
- ricerca:
  - la risposta a ogni istanza è una stringa di bit, che rappresenta un'opportuna informazione.
  - Esempio: dati n dati distinti, identificare tra le n! permutazioni quella che soddisfa una relazione d'ordine (< o >). Si tratta del problema dell'ordinamento.

- verifica:
  - data un'istanza e una soluzione presunta (certificato), si appura se il certificato è davvero una soluzione per l'istanza.
  - Esempio: data una funzione booleana *f* e un'assegnazione di valori alle variabili, si verifica che *f* valga 1 per tale assegnazione
- selezione:
  - caso particolare di verifica: date soluzioni e criterio di accettazione, si separano le soluzioni che soddisfano/non soddisfano il criterio
  - Esempio: dati i risultati di un gruppo di studenti, identificare tutti e soli gli studenti il cui risultato è sopra la media

- simulazione:
  - rappresentazione interattiva della realtà basata sulla costruzione di un modello computazionale di un sistema del quale si vuole analizzare il funzionamento.
  - Esempio: dati n sportelli e un'ipotesi di arrivo di clienti, ciascuno con un servizio di durata temporale nota, prevedere i tempi di attesa
- (ottimizzazione:)
  - data un'istanza e una funzione di costo o vantaggio, tra più soluzioni possibili si seleziona quella a costo minimo o a vantaggio massimo.
  - Esempio: dato contenitore con capacità, dati oggetti con volume e valore, trovare l'insieme di oggetti compatibili con la capacità di valore complessivo massimo (problema del ladro e dello zaino).

Altri criteri di classificazione

- In base all’ambito applicativo: problemi
  - numerici
  - matematici
  - elaborazione di testi
  - elaborazione di informazioni non numeriche
  - etc.
- In base alla natura dei dati primitivi:
  - scalari e/o aggregati
  - vettoriali

- In base ai tipi di dato:
  - numerici (interi o reali)
  - carattere (caratteri o stringhe)
  - tipi di dato astratto
- In base ai costrutti del linguaggio usati:
  - elementari (costrutti condizionali, costrutti iterativi semplici o annidati)
  - avanzati e/o funzioni.

Passi per la soluzione

- identificazione non ambigua e completa del problema da risolvere con analisi ed eventuale completamento delle specifiche
- costruzione di un modello formale del problema
- definizione di un algoritmo di risoluzione e analisi di complessità
- codifica dell'algoritmo in un linguaggio di programmazione
- validazione dell'algoritmo e della sua implementazione su istanze significative.

Algoritmo = strategia

- Scegliere un algoritmo spesso significa individuare la miglior strategia per risolvere un problema
- La scelta si basa su:
  - conoscenza delle operazioni elementari e delle funzioni di libreria fornite dal linguaggio C
  - conoscenza dei costrutti linguistici (tipi di dato, costrutti condizionali e iterativi)
  - conoscenza di algoritmi noti in letteratura
  - esperienza su tipi diversi di problemi.

Strategia

- La maggioranza dei problemi risolti mediante programmi consiste nell'elaborazione di informazioni ricevute in input, per produrre risultati in output
- La parte più rilevante è l'elaborazione, che richiede:
  - l'individuazione di dati (risultati intermedi)
    - che possono essere scalari e/o aggregati
  - la formalizzazione di passi (operazioni) necessarie a valutare i risultati intermedi (a partire da altri dati):
    - i passi intermedi sono spesso formalizzati in termini di costrutti condizionali e/o iterativi. Possono poi essere modularizzati mediante funzioni.

In pratica!

- L'esperienza è un requisito fondamentale (come in molte altre discipline) per una efficace scelta di strategie risolutive: un algoritmo (un programma) è in definitiva un progetto
- Lo studio di problemi classici già risolti è un buon passo di partenza:
  - atteggiamento **errato**: illudersi di risolvere un problema come se si ponesse per la prima volta, senza conoscere quanto già scoperto
  - atteggiamento **corretto**: conoscere e capire la teoria sottostante per poi applicarla.

- Uso di materiale disponibile (in Rete):
  - atteggiamento **errato**: «scopiazzare» senza capire
  - atteggiamento **corretto**: cimentarsi con un problema, poi confrontarsi.
- Talvolta, in mancanza di altri strumenti, un valido punto di partenza è l’analisi della soluzione “a mano” o “su carta” del problema, facendo corrispondere carta (o lavagna) a variabili, calcoli ed espressioni.

Struttura dati

- La scelta della struttura dati deve tener conto:
  - della natura del problema, delle informazioni ricevute in input, dei risultati richiesti
  - delle scelte algoritmiche
- Scelta di una struttura dati = decidere quali (di che tipo) e quante variabili saranno necessarie per immagazzinare dati in input, intermedi e risultati
- Talvolta la struttura dati può essere scelta prima di definire l’algoritmo, ma spesso è necessario considerare dati e algoritmo insieme.

Scelta della struttura dati

- Consiste in:
  - individuare i tipi di informazioni da rappresentare (input, dati intermedi, risultati): numeri (interi o reali), caratteri o stringhe, `struct`
  - decidere se è necessario collezionare i dati in vettori o matrici (aggregati numerabili) oppure se sono sufficienti dati scalari (o `struct`)
- Alcuni problemi si risolvono con poche istruzioni (con costrutti condizionali) su dati scalari
- Molti problemi richiedono iterazioni su dati.

Problemi iterativi e vettori

- Un problema iterativo non richiede un vettore quando è sufficiente manipolare il generico (l'i-esimo) dato, senza **"ricordare"** i precedenti. Bastano dati scalari o aggregati.
  - Esempio: sommatoria, ricerca del massimo
- Un problema iterativo richiede un vettore se è necessario raccogliere/collezionare (in variabili) tutti i dati, prima di poterli manipolare. Servono dati vettoriali.
  - Esempio: input di dati, output in ordine inverso

Scelta dell’algoritmo

- Individuare l’algoritmo (costrutti condizionali e/o iterativi) può essere decisamente semplice (suggerito direttamente dal problema)
  - Esempio : problemi numerici/matematici
- In altri casi scegliere un algoritmo consiste nel realizzare un vero progetto, confrontando strategie diverse, valutando pro e contro (vantaggi e costi)
  - Esempio: pianificazione di attività in funzione di criteri di ottimalità.