Elio Giovannetti
Dipartimento di Informatica
Università di Torino

PROGRAMMAZIONE 2
CORSO DI DIPLOMA

LUCIDI 2000

DIMOSTRAZIONI PER INDUZIONE

Se, data un'asserzione P sui numeri naturali:

dimostro che P vale per il numero 0

assumendo che P valga per un naturale k, riesco a dimostrare che allora vale per k+1

allora ho dimostrato che P vale per ogni n

Lo schema generale della dimostrazione è quindi il seguente:

DIMOSTRAZIONE PER INDUZIONE DI P(n)

BASE DELL'INDUZIONE: P(0)

dimostrazione dell'asserzione P(0)

PASSO DI INDUZIONE

IPOTESI INDUTTIVA: P(k)

TESI INDUTTIVA: P(k+1)

dimostrazione dell'asserzione P(k+1) usando l'ipotesi P(k)

FINE DELLA DIMOSTRAZIONE

Nota: Di solito (ma non sempre) la dimostrazione della base è ovvia, mentre il difficile è la dimostr. del passo

ESEMPIO DI DIMOSTRAZIONE PER INDUZIONE:

LA DIMOSTRAZIONE PER INDUZIONE DELLA FORMULA

$$1+2+\dots+n=\frac{n(n+1)}{2}$$

per qualunque $n \geq 1$

BASE ($n=1$)

Tesi $1=\frac{1(1+1)}{2}$

Dimostrazione: ovvia perchè $\frac{1(1+1)}{2}=\frac{2}{2}=1$

PASSO INDUTTIVO (o PASSO DI INDUZIONE)

IPOTESI INDUTTIVA:

$$1+2+\dots+k=\frac{k(k+1)}{2}$$

TESI INDUTTIVA

$$1+2+\dots+(k+1)=\frac{(k+1)((k+1)+1)}{2}$$

cioè

$$1+2+\dots+(k+1)=\frac{(k+1)(k+2)}{2}$$

Dimostrazione:

$$1+2+\dots+(k+1)$$

$$=\underbrace{(1+2+\dots+k)}_{\downarrow \text{ per ip. induttiva}}+(k+1)$$

$$=\frac{k(k+1)}{2}+(k+1)=\frac{k(k+1)+2(k+1)}{2}=\frac{(k+2)(k+1)}{2}$$

NOTA: La formula precedente può anche essere espressa nella forma iniziante da 0, cioè

$$0+1+2+\cdots+n = \frac{n(n+1)}{2}$$

ossia

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

per qualunque naturale n (anche zero

La dimostrazione è la stessa, ma con base 0 invece di 1

BASE ($n=0$)

Tesi: $0 = \frac{0(0+1)}{2}$

Dimostrazione: ovvio, perché $\frac{0(0+1)}{2} = \frac{0 \cdot 1}{2} = 0$

PASSO D'INDUZIONE

(come prima)

ESERCIZIO: Dimostrare per induzione la formula

$$1^2+2^2+\cdots+n^2 = \frac{2n^3+3n^2+n}{6}$$

INDUZIONE E ITERAZIONE

UN ESEMPIO

Il problema della bandiera tricolore

Dato un vettore *a* di elementi ordinabili (ad es. un vettore di interi), e dati due valori *x1* ed *x2*, con *x1<x2*, si ordini parzialmente il vettore *a* in modo che tutti gli eventuali elementi minori o uguali ad *x1* precedano tutti gli eventuali elementi compresi fra *x1* ed *x2*, i quali a loro volta precedano tutti gli eventuali elementi maggiori di *x2*.

Più precisamente: in modo che esistano due indici *i* e *j* (non necessariamente distinti) tali che:

$x \in a[1..i-1] \rightarrow x \leq x1$
$x \in a[i..j-1] \rightarrow x1 < x \leq x2$
$x \in a[j..n] \rightarrow x > x2$

Nota Bene: non si richiede che il vettore venga totalmente ordinato.

Altra formulazione: Dato un vettore di elementi colorati in uno dei tre colori della bandiera (dove elementi di uno stesso colore possono essere fra loro diversi per altre caratteristiche), si ordini parzialmente il vettore in modo che tutti gli eventuali elementi verdi precedano tutti gli eventuali elementi bianchi i quali a loro volta precedano tutti gli eventuali elementi rossi. Cioè:

SITUAZIONE FINALE:

$x \in a[1..i-1] \rightarrow$ x è verde
$x \in a[i..j-1] \rightarrow$ x è bianco
$x \in a[j..n] \rightarrow$ x è rosso

(dove $1 \leq i \leq j \leq n+1$)

Come si risolve il problema.

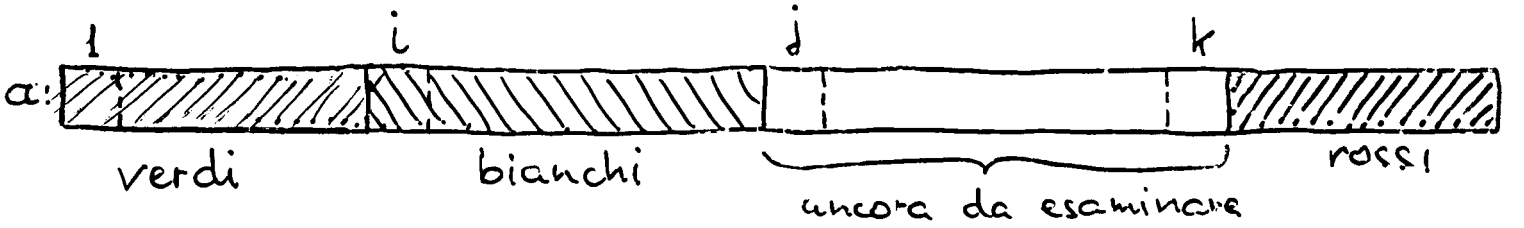
Consideriamo la situazione al passo generico: una parte del vettore sarà già stata esaminata e si avranno quindi tre segmenti rispettivamente verde, bianco, rosso; una parte sarà ancora da esaminare. Il vettore sarà quindi diviso in 4 parti: sono pertanto necessari 3 indici: i,j,k.

È conveniente tenere la parte verde all'inizio del vettore (cioè da 1 a i-1) e la parte rossa alla fine (cioè da k+1 a n);
la parte bianca si può scegliere di tenerla adiacente al verde oppure al rosso. Scegliamo ad esempio la prima possibilità.

Ora scriviamo le istruzioni del corpo del ciclo in modo da diminuire di 1 la lunghezza della parte non esaminata, aggiornando la partizione del vettore.

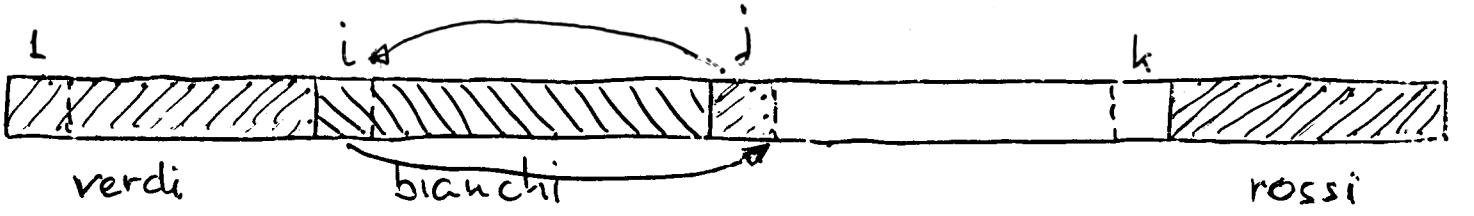
Scriviamo poi le istruzioni di inizializzazione, in modo che inizialmente le tre parti colorate siano vuote, e il test del ciclo, in modo da terminare quando la parte da esaminare diventa vuota.

IL PROBLEMA DELLA BANDIERA



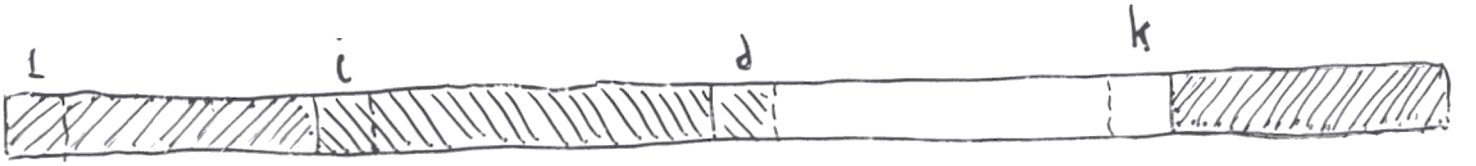
Esaminiamo a[j].

caso 1 : a[j] è verde



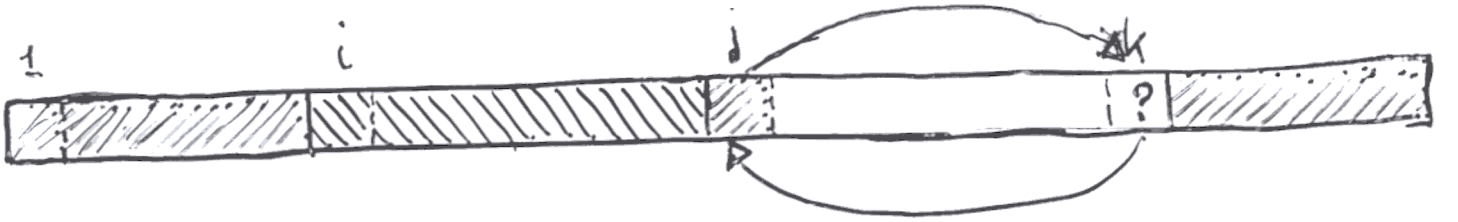
scambia(a[i] a[j]); inc(i); inc(j)

caso 2 : a[j] è bianco



inc(j)

caso 3 : a[j] è rosso



scambia(a[j], a[k]); dec(k)

CONDIZIONE DI USCITA $\neg B$ il segmento da esaminare è vuoto, cioè

$j = k+1 \quad (j > k)$

```
const n = ...
type italcolore = (verde, bianco, rosso);
     elemcolorato = record
                      nome: ...;
                      colore: italcolore;
                    end;

     bandiera = array[1..n] of elemcolorato;

procedure dividicolori(var a: bandiera);
var i,j,k: integer;
begin
  i:= 1; j:= 1; k:= n;
  while j<=k do
    if a[j].colore = verde then begin
      scambia(a[i], a[j]);
      inc(i); inc(j)
    end
    else if a[j].colore = bianco then inc(j)
    else {a[j].colore = rosso} begin
      scambia(a[j], a[k]);
      dec(k)
    end
end;
```

Dimostrazione di correttezza del programma.

INVARIANTE DEL CICLO

Se x è un elemento del pezzo di vettore a[1..i-1], allora x è verde,

se x è un elemento del pezzo di vettore a[i..j-1], allora x è bianco,

se x è un elemento del pezzo di vettore a[k+1..n], allora x è rosso,

(quindi a[j..k] è la parte ancora da esaminare)

dove inoltre: $i \leq n+1$, $j \leq k+1$, $k \leq n$.

In formula logica:

INV:

$$
\begin{aligned}
& i \leq n+1 \ \wedge \ j \leq k+1 \ \wedge \ k \leq n \\
\wedge \ & x \in a[1..i-1] \ \rightarrow \ x \text{ è verde} \\
\wedge \ & x \in a[i..j-1] \ \rightarrow \ x \text{ è bianco} \\
\wedge \ & x \in a[k+1..n] \ \rightarrow \ x \text{ è rosso}
\end{aligned}
$$

L'invariante è vero subito dopo l'inizializzazione, cioè immediatamente prima di eseguire per la prima volta il corpo del ciclo;
esso si mantiene vero dopo ogni iterazione del ciclo;
esso è quindi vero all'uscita del ciclo.

Più precisamente, si ha il seguente teorema:

Teorema 1.
INV è vero dopo qualunque numero N (anche 0) di iterazioni del ciclo.

Traccia di dimostrazione per induzione.

Base:
INV è vero dopo 0 iterazioni, cioè subito prima di eseguire per la prima volta il ciclo.
Dimostrazione della Base:
Inizialmente si ha i=1, j=1, k=n, quindi i tre segmenti di vettore sono vuoti, quindi INV è banalmente vero.

Passo Induttivo:

Ipotesi:
INV sia vero dopo un numero H di iterazioni e sia j<=k.

Tesi:
INV è ancora vero dopo la H+1-esima iterazione.

Dimostrazione del passo:
Basta osservare il codice del corpo del ciclo: si vede che, alla fine della sua esecuzione, i valori di i, j, k saranno in generale cambiati, ma la situazione descritta dall'invariante si sarà riprodotta con tali nuovi valori.

Fine della dimostrazione del teorema.

Poiché INV è vero dopo qualunque numero di ripetizioni del ciclo, esso è vero in particolare all'uscita dal ciclo.
Ma all' uscita del ciclo deve essere vera anche la negazione del test (altrimenti non si sarebbe usciti!).
Deve quindi essere vero:

$$\mathbf{INV} \wedge \neg \mathbf{TEST}$$

cioè

$$\mathbf{INV} \wedge j>k$$

Si vede facilmente che deve essere in particolare:

$$j = k+1$$

Allora si ha

$$\mathbf{INV} \wedge j=k+1$$

Si può quindi in INV sostituire k con j-1, e si ottiene così proprio la SITUAZIONE FINALE voluta.

Utilizzando il Teorema 1 abbiamo così dimostrato il

Teorema 2.
All'uscita dal ciclo il vettore è completamente ripartito in tre segmenti contigui, rispettivamente verde, bianco, rosso, cioè:

SITUAZIONE FINALE:

$x \in a[1..i-1] \rightarrow$ x è verde
$x \in a[i..j-1] \rightarrow$ x è bianco
$x \in a[j..n] \rightarrow$ x è rosso
(dove $1 \le i \le j \le n+1$)

INTRODUZIONE ALLA RICORSIONE

Procedura o funzione ricorsiva:

procedura o funzione che contiene nel suo corpo una chiamata a se stessa (ricorsione diretta)

Esempio:

```
procedure p(n: integer);
var m: integer;
begin
   ...
   p(m);
   ...
end;
```

Procedure o funzioni mutuamente ricorsive:

insieme di procedure o funzioni tali che il grafo della relazione "chiama" è ciclico

Esempio:
P chiama Q, Q chiama R, R chiama P

```
function r(s: string): integer;
var m: integer;
begin
  ...
  p(m);
  ...
end;

procedure q(n: integer);
var k: integer;
    str: string;
begin
  ...
  k:= r(str);
  ...
end;

procedure p(n: integer);
begin
  ...
  q(...);
  ...
end;
```

Realizzazione ricorsiva del fattoriale

definizione: $n! = 1 \cdot 2 \cdot \cdots \cdot n$

si ha:
$1! = 1$
$n! = (n-1)! \cdot n$ per $n>1$

```
function fact(n: integer): longint;
begin
  if n<=1 then fact:= 1
  else fact:= n*fact(n-1)
end;
```

Teorema (di correttezza).
Per qualunque $N \geq 1$, la funzione Pascal *fact*, invocata con un argomento di valore N, restituisce il valore di N!.

Dimostrazione per induzione.
Base
Se N=1, *fact* restituisce 1.
Dimostrazione della base:
ovvia, osservando il testo della funzione.

Passo
Ipotesi: fact(M-1) restituisce (M-1)!
Tesi: *fact(M)* restituisce M!
Dimostraz. del passo: ovvia, osservando ...

Valutazione della complessità temporale delle procedure ricorsive: introduzione.

Complessità temporale del fattoriale ricorsivo.

Sia

T(N) = tempo impiegato per eseguire *fact(N)*

allora si ha (equazioni di ricorrenza):

(1) $T(1) = T_1$
(2) $T(m) = T_2 + T(m-1)$

dove:

T_1 è il tempo occorrente per eseguire il test, prendere il ramo *then* e restituire il risultato;

T_2 è la somma dei tempi impiegati prima e dopo la chiamata ricorsiva, cioè
prima: per eseguire il test, prendere il ramo *else*, fare la sottrazione m-1;
dopo: per fare la moltiplicazione e restituire il risultato.

Soluzione delle equazioni di ricorrenza

(1) $T(1) = T_1$
(2) $T(m) = T_2 + T(m-1)$

Espandiamo T(n) applicando la formula (2):

$T(n) = T_2 + \underline{T(n-1)}$

ora espandiamo T(n-1) riapplicando la (2):
$T(n-1) = T_2 + T(n-2)$
sostituendo si ha:

$T(n) = T_2 + (\underline{T_2 + T(n-2)}) = 2T_2 + \underline{T(n-2)}$

ora espandiamo T(n-2) riapplicando la (2):
$T(n-2) = T_2 + T(n-3)$
sostituendo si ha:

$T(n) = 2T_2 + \underline{T_2 + T(n-3)} = 3T_2 + \underline{T(n-3)}$

...

$T(n) = (n-1)T_2 + T(1) = (n-1)T_2 + T_1$

$T(n) = \Theta(n)$

il tempo di calcolo è <u>lineare in</u> n,
ossia <u>proporzionale a</u> n.

Osserva:

Il valore delle costanti T_1 e T_2 (purchè maggiori di zero) non ha alcuna influenza sull'ordine di infinito; avremmo quindi potuto, per semplificare i calcoli, porle entrambe uguali a 1:

(1) $T(1) = 1$
(2) $T(m) = 1 + T(m-1)$

Il calcolo del lucido precedente diventa:

$T(n) = 1 + \underline{T(n-1)}$

ora espandiamo T(n-1) riapplicando la (2):
T(n-1) = 1 + T(n-2)
sostituendo si ha:

$T(n) = 1 + (\underline{1 + T(n-2)}) = 2 + \underline{T(n-2)}$

ora espandiamo T(n-2) riapplicando la (2):
T(n-2) = 1 + T(n-3)
sostituendo si ha:

$T(n) = 2 + \underline{1 + T(n-3)} = 3 + \underline{T(n-3)}$

...

$T(n) = (n-1) + T(1) = (n-1) + 1 = n$

Complessità spaziale del fattoriale ricorsivo.

S(n) = spazio occorrente per eseguire
il calcolo di fact(n)

= massima profondità dello stack

= proporzionale a n

Confronto fra le versioni iterativa e ricorsiva.

Nel fattoriale iterativo lo spazio occorrente è una costante indipendente da n (poichè viene creato un solo record di attivazione).

Per quanto riguarda il tempo, entrambe le versioni hanno complessità lineare, ma la versione ricorsiva ha di solito una costante di proporzionalità più grande, perchè la chiamata ricorsiva di procedura costa di più della ripetizione del ciclo.

La versione iterativa non è più difficile da scrivere o da capire rispetto alla versione ricorsiva.

Conclusione: la versione iterativa è preferibile.

Realizzazione ricorsiva dell'esponenziale ingenuo

definizione: $\exp(x,n) = x \cdot x \cdot \ldots \cdot x$ *(n volte)*

si ha:
$\exp(x,0) = 1$
$\exp(x,n) = x \cdot \exp(x,n-1)$ per $n>0$

```
function exp(x:real; n:integer):real;
begin
  if n<=0 then exp:= 1
  else exp:= x*exp(x,n-1)
end;
```

Del tutto analogo al fattoriale:

$T(n) = \Theta(n)$
$S(n) = \Theta(n)$

Anche qui è quindi preferibile la versione iterativa.

UN PROBLEMA CON SOLUZIONE RICORSIVA: GENERAZIONE DELLE PERMUTAZIONI

Si definisca una procedura

```
procedure permuta(var v: vettore; n: integer);
```

la quale generi una dopo l'altra, scrivendole via via sullo schermo, tutte le permutazioni degli n elementi del vettore v.

Come si trova la soluzione.

Siano

$P1 = x_{i+1}, x_{i+2}, \ldots, x_{n-1}, x_n$

$P2 = x_{i+1}, x_{i+2}, \ldots, x_n, x_{n-1}$

...

$Pz = x_n, x_{n-1}, \ldots, x_{i+1}, x_i$

tutte le permutazioni della sequenza $x_{i+1}, x_{i+2}, \ldots, x_{n-1}, x_n$

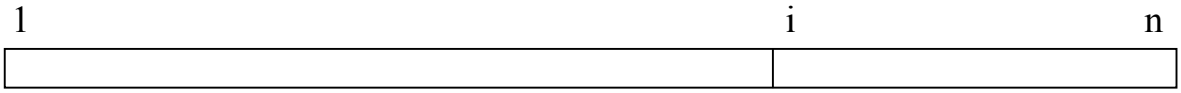
Le permutazioni della sequenza $\mathbf{x_i}, x_{i+1}, x_{i+2}, \ldots, x_{n-1}, x_n$ sono allora:

$\mathbf{x_i}$, P1;	$\mathbf{x_{i+1}}$, P1[$\mathbf{x_i}$ al posto di $\mathbf{x_{i+1}}$];	$\mathbf{x_{i+2}}$, P1[$\mathbf{x_i}$ al posto di $\mathbf{x_{i+2}}$];	...
$\mathbf{x_i}$, P2;	$\mathbf{x_{i+1}}$, P2[$\mathbf{x_i}$ al posto di $\mathbf{x_{i+1}}$];	$\mathbf{x_{i+2}}$, P2[$\mathbf{x_i}$ al posto di $\mathbf{x_{i+2}}$];	...
...	...	...	
$\mathbf{x_i}$,Pz;	$\mathbf{x_{i+1}}$, Pz[$\mathbf{x_i}$ al posto di $\mathbf{x_{i+1}}$];	$\mathbf{x_{i+2}}$, Pz[$\mathbf{x_i}$ al posto di $\mathbf{x_{i+2}}$];	...

Definiamo una procedura ricorsiva

`procedure perm(i: integer)`;

che produca nel sottovettore v[i..n] successivamente tutte le permutazioni degli elementi del sottovettore stesso e per ognuna di tali permutazioni scriva sullo schermo l'intero vettore. Inoltre la procedura riporti alla fine il sottovettore v[i..n] nella permutazione di partenza.



Caso base: i = n

Il sottovettore v[i..n] contiene un solo elemento, quindi la permutazione è una sola, la procedura non deve effettuare alcuna modifica del vettore ma soltanto visualizzare l'intero vettore:

`if i=n then scrivivettore(v,n)`

Caso ricorsivo: i < n

Il sottovettore v[i..n] contiene più di un elemento.

Assumiamo (*ipotesi induttiva*) che **`perm(i+1)`** produca tutte le permutazioni di v[i+1..n] e per ognuna di esse scriva l'intero vettore, **riportando alla fine v[i+1..n] nella situazione precedente la chiamata**.

Allora per ottenere tutte le permutazioni di v[i..n], riportando alla fine v[i..n] nella situazione precedente la chiamata (cioè per fare in modo che la *tesi induttiva* sia dimostrabile) bisogna:

1) produrre tutte le permutazioni di v[i+1..n] lasciando fisso v[i];

2) scambiare v[i] con v[i+1] e poi produrre tutte le permutazioni del nuovo v[i+1..n] lasciando fisso (il nuovo) v[i], infine ripristinare i precedenti v[i] e v[i+1];

3) scambiare v[i] con v[i+2] e poi produrre tutte le permutazioni del nuovo v[i+1..n] lasciando fisso (il nuovo) v[i], infine ripristinare i precedenti v[i] e v[i+2];

4) scambiare v[i] con v[i+3] e poi produrre tutte le permutazioni del nuovo v[i+1..n] lasciando fisso (il nuovo) v[i], infine ripristinare i precedenti v[i] e v[i+3];

...

n-i+1) scambiare v[i] con v[n] e poi ecc.

cioè:

```
for k:= i to n do begin
  scambia(v[k],v[i]);
  perm(i+1);
  scambia(v[k],v[i])
end
```

Racchiudiamo la definizione della procedura ricorsiva **perm** all'interno di una procedura non ricorsiva avente come parametri il vettore **v** e la sua lunghezza "effettiva" **n**; in questo modo la procedura interna perm può accedere a **v** ed **n** senza che occorra passarli invariati ad ogni chiamata ricorsiva:

```
procedure permuta(var v:vettore; n:integer);

  procedure perm(i:integer);
  var k: integer;
  begin
    if i=n then scrivivettore(v,n)
    else
      for k:= i to n do begin
        scambia(v[k],v[i]);
        perm(i+1);
        scambia(v[k],v[i])
      end
  end;

begin{permuta}
  perm(1)
end;
```