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.
<u>*Base*</u>
Se N=1, *fact* restituisce 1.
Dimostrazione della base:
ovvia, osservando il testo della funzione.

<u>*Passo*</u>

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 lineare in n,
ossia proporzionale a 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.

<u>Conclusione: la versione iterativa è preferibile</u>.

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

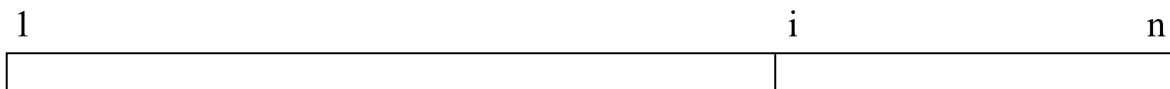
Le permutazioni della sequenza $\mathbf{x_i}, x_{i+1}, x_{i+2}, \ldots, x_{n-1}, x_n$ sono allora:

$\mathbf{x_i}$, 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. <u>Inoltre la procedura riporti alla fine il sottovettore v[i..n] nella permutazione di partenza.</u>



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;
```

Complessità temporale

Ricordiamo che il numero di tutte le possibili permutazioni di *n* elementi è *n!*: infatti il primo elemento può essere scelto in n modi, per ognuna di tali scelte il secondo elemento può essere scelto in (n-1) modi, producendo così n(n-1) scelte; per ognuna di esse il terzo elemento può essere scelto in (n-2) modi, e così via: $n(n-1)(n-2) \ldots 2 \cdot 1$

Qualunque procedura che generi tutte le permutazioni di n elementi dovrà quindi eseguire (almeno) un numero di passi proporzionale a n!.

Verifichiamo per mezzo di una relazione di ricorrenza.

Il tempo occorrente per calcolare l'unica permutazione di un elemento è ovviamente un tempo costante (si fa il test, e poi non si fa niente, o meglio si fa solo l'output):

$$T(1) = 1$$

Il tempo occorrente per calcolare le permutazioni di n elementi è quello occorrente per un ciclo *for* di m iterazioni, in ognuna delle quali si esegue una chiamata ricorsiva che calcola le permutazioni di m-1 elementi:

$$T(m) = m(1+T(m-1)) > mT(m-1)$$

Anbbiamo dunque la seguente relazione di ricorrenza:

$$T(1) = 1$$
$$T(m) > mT(m-1)$$

Espandiamo T(n) per mezzo di essa:

$$T(n) > nT(n-1) > n(n-1)T(n-2) > n(n-1)(n-2)T(n-3) > \ldots$$

$$> n(n-1)(n-2) \ldots 2 \cdot 1 = n!$$

Quindi:

$$T(n) = \Omega(n!)$$

ma $n! = \Theta(n^n)$, quindi $T(n) = \Omega(n^n)$ <u>super-esponenziale</u>

Si osservi tuttavia che, come già nel caso delle torri di Hanoi, non è l'algoritmo che è "cattivo", è il problema che è "cattivo", perché richiede come output la produzione di n! sequenze.

Complessità spaziale

S(n) è data dalla massima profondità dello stack, che è proporzionale a *n*, perchè i record di attivazione creati in iterazioni distinte di un ciclo *for* non sono attivi contemporaneamente: al massimo si hanno *n* attivazioni contemporanee.

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