APPUNTI LEZIONI PROGRAMMAZIONE II
A CURA DELLA PROF.SSA EMANUELA FACHINI

Ricorsione.

Introduzione.

In questi appunti cercheremo di esaminare piu' da vicino la ricorsione, come tecnica di programmazione, confrontando soluzioni ricorsive ed iterative dal punto di vista dell'efficienza e dal punto di vista della semplicità progettuale. Vedremo poi come la ricorsione debba essere vista come una tecnica per il "problem solving", cioè uno strumento per costruire soluzioni di problemi.
Lo svantaggio delle funzioni definite ricorsivamente risiede nel maggiore dispendio di risorse necessarie per il loro calcolo, ma questo non è sempre vero e inoltre potrebbe dipendere dall'algoritmo e non dall'implementazione ricorsiva!

I vantaggi nell'uso della ricorsione come tecnica di programmazione sono che:

- i programmi ricorsivi sono più chiari, più semplici, più brevi e più facili da capire delle corrispondenti versioni iterative.
- il programma riflette fedelmente la strategia di soluzione del problema.
- spesso la soluzione trovata può poi trasformarsi più o meno meccanicamente in una soluzione iterativa equivalente ma più efficiente.

Nel caso si voglia calcolare qualche cosa su numeri, su vettori o liste concatenate è spesso relativamente facile costruire funzioni C iterative che svolgono il compito voluto, ma su strutture dati non lineari come gli alberi non c'è un modo ovvio per costruire versioni iterative. Vedremo di approfondire questi argomenti nel seguito.

Ricorsione e induzione.

Notiamo che se si definisce induttivamente un insieme viene più naturale definire ricorsivamente funzioni su quell'insieme.
Per esempio prendiamo l'insieme dei numeri naturale e la sua definizione induttiva:

- 0 è un numero naturale
- se n è un numero naturale anche il suo successore è un numero naturale
- niente altro è un numero naturale

In generale ogni insieme definito induttivamente si costruisce così:

- si sceglie un numero finito di elementi come gli elementi di base dell'insieme
- si stabilisce una o più regole per costruire da elementi dell'insieme nuovi elementi dell'insieme, precisando che solo gli elementi così ottenuti fanno parte dell'insieme.

Altro esempio: le stringhe

- la stringa vuota è una stringa
- se s è una stringa, concatenando un carattere a sinistra otteniamo una stringa
- niente altro è una stringa

Alcuni esempi di funzioni definite ricorsivamente.

Esempio 1. Vediamo allora come può essere naturale definire la somma tra due numeri naturali ricorsivamente:

Sum(x,0) = x
Sum(x,y+1) = Sum(x,y) +1

Questa definizione si traduce immediatamente in una funzione C ricorsiva:

```
int sum(int x, int y)
/*prec: x,y≥ 0
postc: restituisce x+y, con x+y ≥0 */
{if (y == 0) return x;
```

```
  return 1 + sum(x,y-1);
}
```

Esempio 2. Analogamente per il prodotto:

```
Prod(x,0)= 0
Prod(x,y+1) = Sum(Prod(x,y),x)
```

funzione C:

```
int prod (int x, int y)
/*prec: x,y≥ 0
postc: restituisce x*y, con x*y≥0  */
{
  if (y == 0) return 0;
  return sum(x,prod(x,y-1));
}
```

Esempio 3. A questo punto possiamo definire ricorsivamente l'elevamento a potenza, basandoci sul prodotto:

```
Pot(b,0) = 1
Pot(b,n+1) = Prod(b,Pot(b,n))
```

E la relativa implementazione in C

```
int pot(int b, int i)
/*prec: b,i≥ 0
postc: restituisce b^i , con b^i≥0*/
{
 if (i == 0) return 1;
 return (prod(b,pot(b,i-1)));
 }
```

Naturalmente per assicurarsi che il processo ricorsivo termini bisogna che l'argomento su cui si ricorre diminuisca a ogni chiamata, inoltre in questo caso vogliamo definire le operazioni solo sui numeri non negativi e quindi sono state introdotte delle precondizioni. Se il chiamante chiama la funzione su dati di input che rispettano la precondizione allora può aspettarsi il risultato descritto dalla postcondizione. In altre parole una precondizione consiste di una o più condizione sui parametri di input che deve essere

garantita dal chiamante, pena un comportamento imprevedibile o scorretto della funzione stessa.

Esempio 4. Consideriamo un altro esempio, vogliamo calcolare il numero di occorrenze di un carattere c in una stringa s. Metodo ricorsivo basato sulla definizione induttiva di stringa:

- se s è la stringa vuota il risultato è 0;
- altrimenti se il primo carattere della stringa coincide con c restituisci 1 + il numero delle occorrenze di c nel resto della stringa s (tolto il primo carattere)
- altrimenti (cioè se il primo carattere di s è diverso da c) restituisci il numero delle occorrenze di c nel resto di s.

La funzione C si ottiene subito, utilizzando l'aritmetica dei puntatori:

```
int occCar(char* s, char c)
/*prec: s!= NULL,
postc: restituisce il numero di occorrenze del carattere in
c nella stringa s*/
{if (*s == '\0') return 0;
  if (*s == c) return 1 + occCar(++s, c);
  else return occCar(++s, c);}
```

Esempio 5. Consideriamo ancora un altro esempio: il fattoriale, la cui definizione ricorsiva è

$n! = 1$ se $n=0$ e
$n! = n*(n-1)!$ per $n \geq 1$

che si traduce nella seguente della funzione C ricorsiva:

```
int fattoriale(int n)
/*prec: n≥0
Postc:restituisce il fattoriale di n */
{if (n==0) return 1;
 return  n * fattoriale(n-1);
}
```

Ricorsione ed efficienza.

Come si valuta l'efficienza di una funzione ricorsiva?

Nell'esecuzione di una funzione ricorsiva si ha la chiamata della funzione stessa più e più volte, mentre alla fine dell'esecuzione di un ciclo semplicemente si torna all'inizio del ciclo stesso.
E' chiaro che una chiamata di funzione è più costosa di un ritorno all'inizio del ciclo, infatti:

per lo spazio

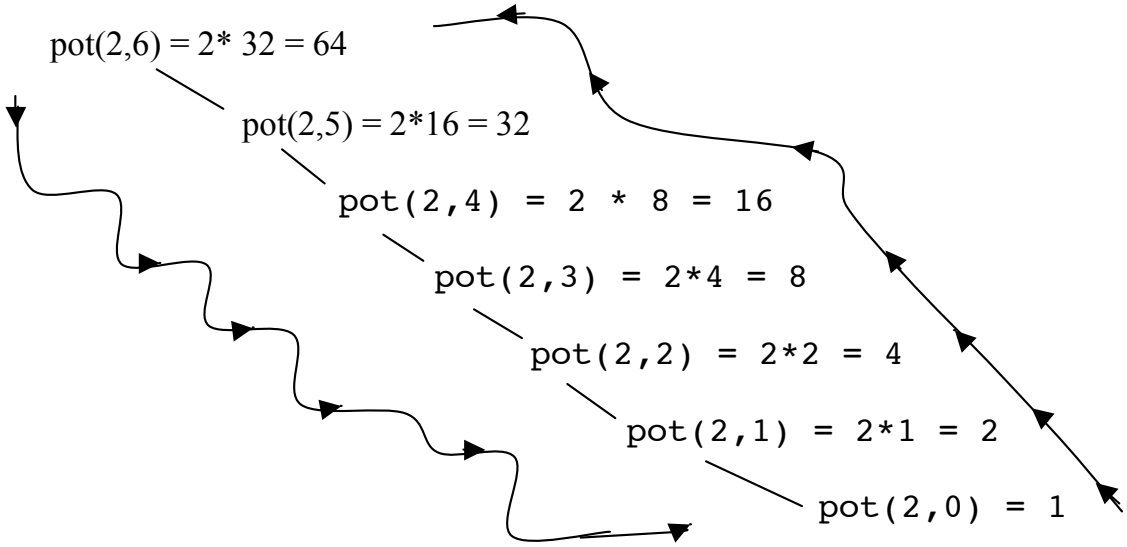
Ogni chiamata della funzione può richiedere spazio per i parametri e le variabili locali, oltre che per l'indicazione del punto di rientro della chiamata. Queste informazioni (record di attivazione) sono memorizzate in una pila dalla quale vengono automaticamente cancellate non appena la chiamata di una funzione è terminata .
Quindi possiamo dire che l'esecuzione di una funzione ricorsiva richiede un'occupazione in spazio almeno proporzionale al numero delle chiamate.

per il tempo

Le operazioni coinvolte in una chiamata (allocazione e rilascio della memoria, copia dei valori dei parametri nella memoria locale, rientro dalla chiamata) contribuiscono tutte ad appesantire il tempo di calcolo.

Analisi efficienza funzioni degli esempi 3 e 5:
Quanto costa calcolare ricorsivamente la potenza n-sima di un numero b?

Consideriamo l'albero delle chiamate per pot(2,6) nella figura seguente:



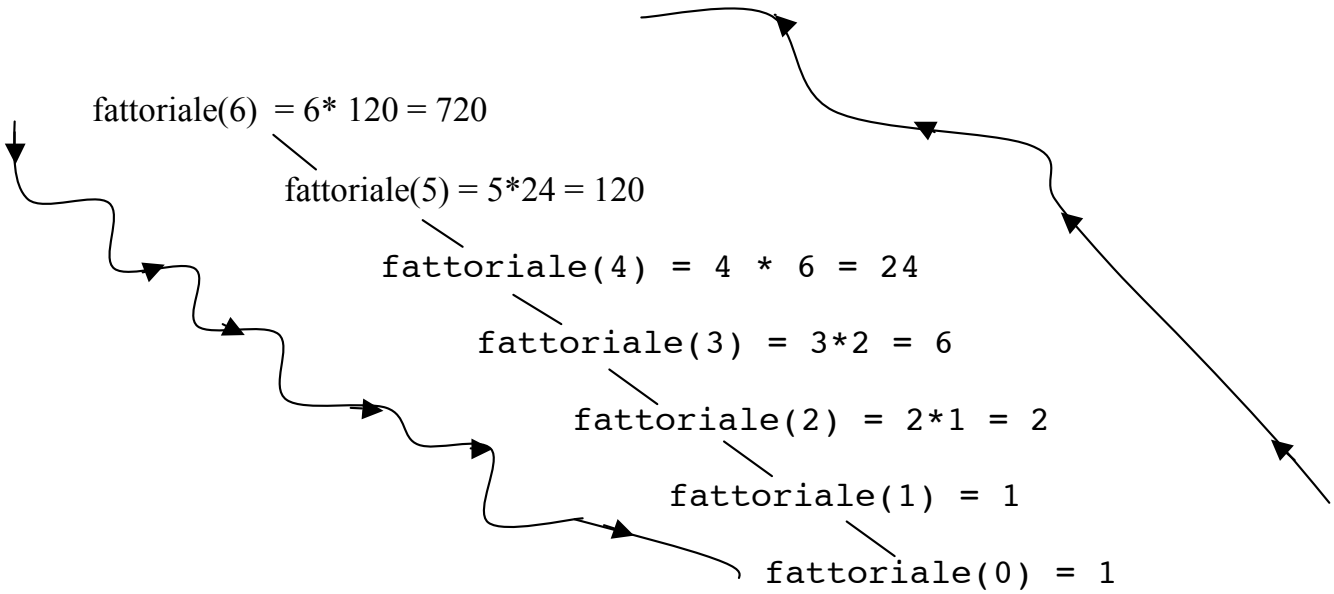
Si calcolano n prodotti e e si occupa uno spazio di memoria proporzionale a n, perché si deve considerare lo spazio per le chiamate in sospeso. La versione iterativa equivalente invece comporta l'esecuzione di n prodotti ma in spazio costante:

```
int potIter(int b, int n)
/*prec: b,n≥ 0
postc: restituisce bⁿ , con bⁿ≥0*/
{int ris = 1;
for (;0<n;n--)
ris = ris*b;
return ris;}
```

Consideriamo la funzione C ricorsiva che calcola il fattoriale:

```
int fattoriale(int n)
/*prec: n≥0
Postc:restituisce il fattoriale di n */
{if (n==0) return 1;
 return  n * fattoriale(n-1);
}
```

Vediamo come procede il calcolo, come prima costruendo l'albero delle chiamate:



Per calcolare il fattoriale di n bisogna eseguire un numero proporzionale a n di prodotti e anche l'occupazione di spazio di memoria è proporzionale a n.

Esempio 6. Consideriamo una altro classico esempio: i numeri di Fibonacci:

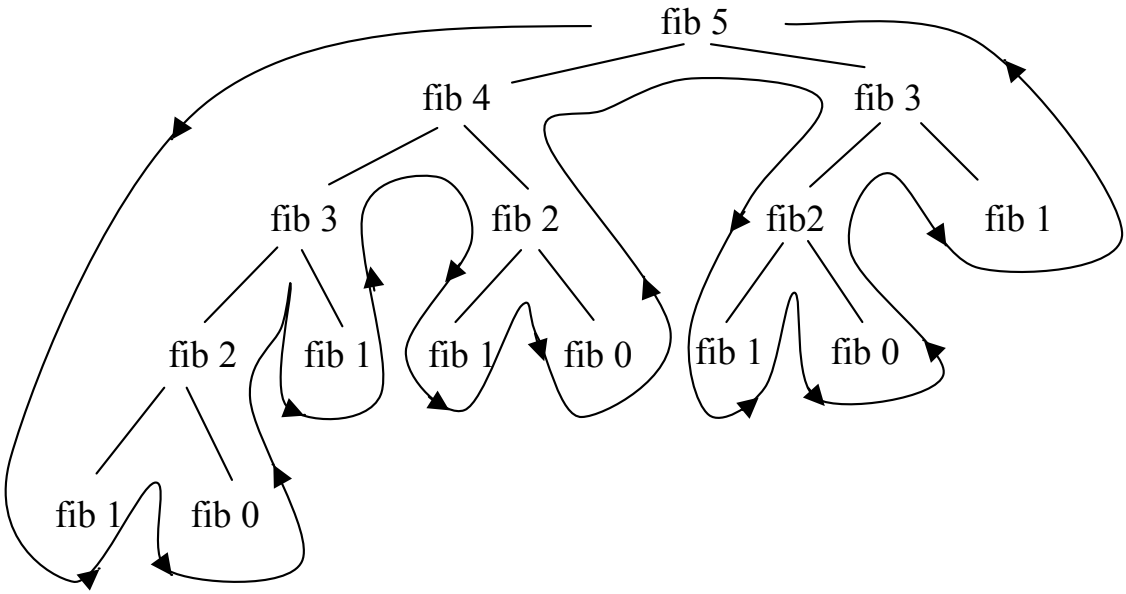
La definizione ricorsiva dei numeri di Fibonacci è:

$$Fib(n) = \begin{cases} 0 \text{ se } n = 0 \\ 1 \text{ se } n = 1 \\ Fib(n-1)+Fib(n-2) \text{ altrimenti} \end{cases}$$

Di nuovo la definizione si può trasferire immediatamente in un calcolo ricorsivo, anche se questa volta si tratta una **ricorsione multipla**:

```
int fib(int n)
/*prec: n≥0
postc: restituisce l'n-simo numero di Fibonacci*/
 { if (n==1 || n==0) return n;
 return fib(n-1)+ fib(n-2)}
```

Questa procedura è istruttiva perchè consente di esemplificare un albero delle chiamate che non è lineare : la funzione chiama se stessa due volte e quindi l'albero è binario.



Esaminiamo più attentamente il calcolo: per calcolare (fib 5), calcoliamo (fib 4) e (fib 3), ma per calcolare (fib 4), calcoliamo (di nuovo!) (fib 3) e (fib 2) e così via. Ma rappresenta un modo terribile di calcolare i numeri di Fibonacci perchè si fanno molti calcoli più di una volta. Per esempio il calcolo di fib 3 è eseguito due volte, ed è quasi la metà di tutto il calcolo!

Non è difficile fare vedere che il numero di volte che la procedura calcola (fib 1) o (fib 0) è proprio (fib n+1), funzione che cresce esponenzialmente con n. Mentre lo spazio richiesto cresce solo linearmente con l'input, infatti è proporzionale all'altezza dell'albero, che fornisce la dimensione massima dello stack delle chiamate.
Quello che è successo con Fibonacci è piuttosto esemplare perchè quando si cerca di risolvere un problema con un approccio ricorsivo si è concentrati sulla correttezza del medoto risolutivo quindi può accadere che la soluzione proposta sia inefficiente. Per questo, dopo aver risolto il problema, è essenziale esaminare criticamente la propria soluzione e se possibile migliorarla, certamente dal punto di vista dell'efficienza, ma anche dal punto di vista dell'eleganza.
Applichiamo questa raccomandazione al caso di Fibonacci.

Osserviamo che per calcolare Fib(n+2) servono Fib(n+1) e Fib(n), e poi per calcolare Fib(n+1) devo calcolare Fib(n) e Fib(n-1), quindi potremmo adottare una soluzione che conserva e "passa" i valori già calcolati.
I valori già calcolati vengono memorizzati in due parametri della funzione per poter essere utilizzati nelle diverse chiamate:

```
int fibon2(int n)
/*postc: se n ≥ 0 restituisce l'n-simo numero di Fibonacci,
-1 altrimenti*/
{int fibn=1,fibnMeno1=0;
if (n < 0) return -1;
ris = fibnMeno1;
if (n > 0) ris = fib2(fibn,fibnMeno1,n);
return ris;}


int fib2(int fibn, int fibnMeno1, int n)
/*prec: n>0
postc: restituisce l'n-simo numero di Fibonacci*/
{if (n==1) return fibn;
return fib2(fibn + fibnMeno1,fibn,n-1);}
```

Con questo secondo metodo si calcola, sempre ricorsivamente, l'n-simo numero di Fibonacci in tempo lineare in n, mentre con il primo metodo si

aveva un numero di passi proporzionale a Fib(n) che cresce esponenzialmente.
Quindi qui il vantaggio è enorme e sensibile anche per piccoli valori di input.
Confrontate la leggibilità delle versioni delle funzioni per il calcolo del fattoriale e dei numeri di Fibonacci.

Le funzioni tail ricursive.

Inoltre fib2, come ogni altra funzione tail recursive, può essere trasformata **a tempo di compilazione** in un'iterativa equivalente. L' approccio ricorsivo è infatti tanto potente ed elegante che si è investito molto nel costruire compilatori in grado di ottimizzarlo e renderlo competitivo con quello iterativo anche dal punto di vista dell'efficienza. Questo si è ottenuto per funzioni in cui l'ultima operazione è la chiamata ricorsiva. Queste funzioni per questo sono dette **tail ricorsive (**ricorsive in coda**)**. Si ottiene così il massimo vantaggio: la semplicità e l'eleganza della funzione per il programmatore e l'efficienza di calcolo. Anche se si deve sottolineare che spesso la versione più "naturale" di una soluzione ricorsiva non è tail ricorsive, benchè non sia difficile, se possibile, trasformare una funzione non tail-ricorsive in una equivalente che lo è. Tipicamente si devono aggiungere dei parametri per accumulare i risultati parziali, come abbiamo fatto per il calcolo dei numeri di Fibonacci.

Vediamo un'altra trasformazione di questo tipo:

Ricordiamo la definizione della funzione per il calcolo del fattoriale:

```
int fattoriale(int n)
/*prec: n≥0
postc:restituisce il fattoriale di n */
{if (n==0) return 1;
 return  n * fattoriale(n-1);
}
```

Notiamo che potremmo evitare di lasciare delle chiamate in sospeso calcolando i prodotti parziali "scendendo" lungo le chiamate e memorizzandoli in un parametro. Tale parametro quindi contiene come primo risultato n, poi n*(n-1), quindi il risultato dell'ulteriore prodotto per n-2, e così via. Il parametro n verrà usato come un contatore e quindi andrà

decrementato a ogni chiamata, il risultato si avrà quando n =1. Questa funzione sarà chiamata da un'altra nella quale si inizializza ris:

```
int fattoriale2(int n)
 /*postc:se n≥0 restituisce il fattoriale di n, altrimenti
0*/
{int ris=1;
if (n < 0) return 0;
if (n > 1)  ris = fattor2(ris,n);
return ris;}

int fattor2(int ris,int n)
/*prec: n > 0
Postc: restituisce ilfattoriale di n */
 {if ( n == 1) return ris;
 return fattor2(n*ris,n-1);}
}
```

La nuova funzione fattoriale2 è stata anche resa più robusta, infatti ora vengono trattati tutti i casi.
La chiamata di fattor2(6) dà origine al seguente calcolo

ris = 6 e n=6
ris = 30, n=5
ris = 120, n=4
ris = 360, n=3
ris = 720, n=2

Vediamo che il comportamento della funzione è molto più vicino a quello di una funzione definita iterativamente. Se confrontiamo i processi generati da fattoriale e da fattoriale2, osserviamo che nel primo caso le chiamate in sospeso nella pila delle chiamate devono essere risolte una alla volta, perchè il risultato deve essere calcolato in ogni passo e restituito al chiamante, mentre nel secondo il risultato è calcolato nell'ultima chiamata così il rientro può portare direttamente alla prima chiamata. Se il conpilatore è in grado di ottimizzare l'implementazione di una funzione tail ricursive questa sarà calcolata in tempo lineare in n e in spazio costante come la seguente versione iterativa equivalente:

```
int fattIterat(int n)
/*postc: se n ≥ 0 restituisce il fattoriale di n, 1
altrimenti */
 {int ris=1;
 while (2 <= n)
 ris = ris * (n--);
 return ris;}
```

Concludiamo notando che il calcolo ricorsivo del fattoriale si può migliorare ancora utilizzando un parametro passato per riferimento, in questo modo evitando che sia prodotta in ogni chiamata la copia locale del valore da restituire, come segue:

```
int fattoriale3(int n)
/*postc: se n ≥ 0 restituisce il fattoriale di n, 0
altrimenti */
{int ris=1;
if (n < 0) return 0;
if (n > 1) fattor3(&ris,n);
return ris;}

void fattor3(int *ris,int n)
/*prec n > 1
Postc:restituisce il fattoriale di n */
{if ( n == 1) return;
*ris = n * (*ris);
fattor3(ris,--n);
}
```

Riprendiamo anche l'esempio 4:

```
int occCar(char* s, char c)
/*prec: s!= NULL,
postc: restituisce il numero di occorrenze del carattere in
c nella stringa s*/
{if (*s == '\0') return 0;
 if (*s == c) return 1 + occCar(++s, c);
 else return occCar(++s, c);}
```

Anche in questo caso non è difficile costruire una funzione tail ricorsive equivalente:

```
int occCar2(char* s, char c)
/*postc: restituisce il numero di occorrenze del carattere
in c nella stringa s, se s != NULL, -1 altrimenti*/
{int ris=0;
if (s) ris = oCar(s,c,ris); else return -1;
return ris;}

int oCar(char* s, char c,int  ris)
/*prec: s!= NULL,
postc: restituisce il numero di occorrenze del carattere in
c nella stringa s*/
 {if (*s == '\0') return ris;
  if (*s == c) return oCar(++s, c,ris+1);
  else return oCar(++s, c,ris);
 }
```

Questa può a sua volta essere resa più efficiente utilizzando il passaggio per riferimento del parametro nel quale calcoliamo il risultato:

```
int occCar3(char* s, char c)
/*postc: restituisce il numero di occorrenze del carattere
in c nella stringa s, se s != NULL, -1 altrimenti*/
{int ris=0;
 if (s)  oCar2(s,c,&ris); else return -1;
 return ris;}

void oCar2(char* s, char c,int * ris)
/*prec: s!= NULL,
postc: restituisce il numero di occorrenze del carattere in
c nella stringa s*/
{if (*s != '\0')
 if (*s == c)
    {(*ris)++;
    oCar2(++s, c,ris);}
 else oCar2(++s, c,ris);
```

```
}
```

Quindi possiamo osservare che non si può dire che la ricorsione sia inefficiente, ma piuttosto che prima di implementare così com'è un algoritmo ricorsivo, bisogna analizzarne l'efficienza e implementarlo come tale se la valutazione è positiva o cercare di trasformarlo in modo che lo diventi o adottando un approccio iterativo o modificando opportunamente la ricorsione.

Ancora un esempio: riconoscere le parole palindrome.

Vogliamo verificare se una stringa è una parola palindroma, cioè può essere letta indifferentemente da destra o da sinistra.
Diamo una definizione ricorsiva di parola parola palindroma:

- la parola vuota è una palindroma
- la parola axb è una palindroma se a=b e x è una palindroma.

Qui la traduzione in C non è così diretta perchè bisogna accedere all'ultimo simbolo della stringa per confrontarlo con il primo, si introduce quindi un parametro ausiliario nella funzione ricorsiva che è inizializzato alla lunghezza della stringa data:

```
int palRic(char* s)
 /*postc: se s != NULL restituisce un valore diverso da 0
se s è una parola palindroma e 0 altrimenti, se s == NULL
restituisce -1 */
{int len;
if (s) {len = strlen(s); return palRicAus(s,len);}
else return -1;}

int palRicAus(char *s, int len )
/*prec: s!= NULL
postc: restituisce un valore diverso da 0 se s è una parola
palindroma, 0 altrimenti */
 {if (len == 0 || len == 1) return 1;
    return (s[0] == s[len-1] && palRicAus(s+1,len-2));
 }
```

Questa funzione è tail ricorsive.

L'iterazione come caso particolare di ricorsione.

Per rendersi conto di come mai una funzione tail ricorsive può essere trasformata a tempo di compilazione in una iterativa equivalente osserviamo il formato generale di una funzione tail ricorsive, nel caso più semplice in cui non ci sia restituzione di valore sarà del tipo:

```
void funzioneTR () { ... ; if (e) funzioneTR(); }
```

che il compilatore può riconoscere e trasformare nell'equivalente

```
void funzioneTR () { start: ...; if (e) goto start; }
```

eliminando quindi l'aumento di occupazione di memoria determinato dallo stack di ricorsione.

Non sempre però le funzioni sono tail ricorsive, o possono essere trasformate facilmente in funzioni equivalenti tail ricorsive.

Consideriamo il problema di stampare una stringa all'inverso, dobbiamo "raggiungere" l'ultimo simbolo della stringa stamparlo, poi il penultimo…, etc. Usando la ricorsione otteniamo una semplice funzione C:

```
void StampaStrInv(char *s)
/*prec s!=NULL
Post: stampa la stringa in input all'inverso */*/
{if (*s == '\0') return;
 StampaStrInv(s+1);
 putchar(*s);}
```

Notiamo che, a differenza degli esempi fin qui descritti, l'ultima istruzione non è la chiamata ricorsiva. Qui l'operazione fatta per ultima **deve** essere la stampa a video del carattere, proprio perché svuotando lo stack delle chiamate ci ritroviamo con i caratteri della stringa in input invertiti.
Compiamo un numero di passi e abbiamo un'occupazione in spazio, lo stack delle chiamate, proporzionale alla lunghezza della stringa in input.
Scrivere una versione iterativa di questa funzione comporta la gestione esplicita di una struttura d'appoggio, come per esempio:

```
void StampaStrInvIter(char *s)
/*prec s!=NULL
Post: stampa la stringa in input all'inverso */
{char *v=s;
while ( *s != '\0') s++;
do
{s--;
putchar(*s);}
while (s != v);
}
```

L'occupazione in spazio è costante, il numero dei passi non cambia, ma si perde in eleganza e semplicità!