DISPENSE

del corso di Programmazione II
Docente: Paolo Terenziani
a.a. 2014-2015

PRIMA PARTE

Le dispense nel seguito devono essere integrate con il manuale del C nel testo: Paul Deitel, Harvey Deitel "***Il linguaggio C. Fondamenti e tecniche di programmazione".*** Pearson Italia, 2013 (indicato nel seguito come: **TESTO**)

Avvertenza. Sebbene gli algoritmi qui riportati siano stati verificati, non si esclude la presenza di errori. Gli studenti che rilevassero eventuali errori sono pregati di segnalarli, al fine di migliorare la qualita' delle note stesse.

Sommario

0. TIPI DEL C: STRUTTURE (“STRUCT”) E PUNTATORI

0.1 Strutture

Da integrare con TESTO, Sezioni 10.1, 10.2, 10.3, 10.4, 10.5, 10.6

Le Strutture sono tipi di dati composti. Nel caso degli array (anche detti **vettori**) si devono dichiarare il tipo di dati che essi contengono (ad esempio, posso avere array di interi, reali, caratteri ….); con le strutture si possono avere più tipi di dati. Vediamo un esempio:

```
struct point {
int x;
int y;
}
```

Nella precedente dichiarazione si utilizza un nuovo tipo di dati: struct, struttura. Esso è caratterizzato da:

- un'etichetta o tag: in questo caso il tag è point. (Così come si fa per le funzioni, è opportuno dare nomi alle strutture in modo tale da poterle riconoscere subito per quello che rappresentano)

- uno o più campi, che servono a memorizzare l'informazione necessaria a rappresentare la struttura in oggetto. Nel nostro esempio la struttura e' un punto, il quale è caratterizzato da un'ascissa ed un'ordinata, in pratica due interi x ed y.

Si noti che, nell’esempio proposto, i campi della struttura sono tutti dello stesso tipo (int). In generale, pero’, i campi di una struttura possono avere tipi diversi, come mostrato dalla struttura “person” nel paragrafo 0.4.

Ora è possibile sfruttare questo nuovo tipo di dati per dichiarare nuove variabili. Supponiamo, ad esempio, di voler avere una variabile p di tipo struttura punto (struct point): ci sono due modi per compiere questa operazione:

```
struct point {
int x;
int y;
}p;
```

```
struct point {
int x;
int y;
};
struct point p;
```

Quindi, nel primo caso si pone il nome del dato di tipo struttura subito dopo la dichiarazione della sua struttura, mentre nel secondo si dichiara p in un momento successivo.

0.2 Utilizzo

Per inserire i dati nella struttura è possibile operare in questo modo:

```
struct point p={10,12};
```

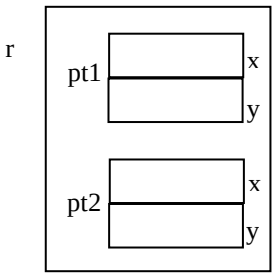
E' necessario inserire tanti valori quanti sono i campi e rispettare i tipi di dati precedentemente dichiarati (cioè se sono stati dichiarati campi di caratteri, non è possibile poi inserire interi).

Per stampare i campi di una struttura si usa un accesso del tipo: nomestruttura.campo:

printf("%d,%d",p.x,p.y);

Così come esistono le singole strutture, allo stesso modo è possibile avere strutture di strutture. Vediamo come esempio un rettangolo ”r” definito da 2 punti, a loro volta strutture.

```
struct rect {
  struct point pt1;
  struct point pt2;
};
struct rect r;
```



Per assegnare alla variabile pippo il valore contenuto nel campo y del primo punto, sarà sufficiente scrivere:

pippo = r.pt1.y = pippo;

Invece, per assegnare direttamente il valore 17 in tale campo, si potra’ usare l’istruzione

r.pt1.y=17;

Vediamo ora come una funzione può restituire una struttura. Costruiamo la funzione “make” che, dati due interi a e b, restituisce una struttura point:

```
struct point make(int a, int b) {
  struct point temp;
  temp.x=a;
  temp.y=b;
  return(temp);
  }
```

Avendo dichiarato "struct point p" sarà ora sufficiente richiamare la funzione make su 2 interi x1 ed x2:

p=make(x1,x2);

In p ora c'è la struttura con l’ascissa (campo x) che assume il valore x1, e l’ordinata (campo y) che assume il valore x2.

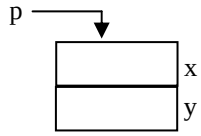
Una funzione che restituisce strutture può, a sua volta, operare su strutture. A questo proposito consideriamo la funzione “addpoint”, che restituisce un nuovo punto, avente come ascissa la somma delle ascisse di p1 e p2, e come ordinata la somma delle ordinate di p1 e p2. Si noti che, poiche’ il C passa (come parametri) le strutture per valore, p1 (e p2) non e’ modificato dalla funzione addpoint.

```
struct point addpoint(struct point p1, struct point p2) {
  p1.x+=p2.x;
  p1.y+=p2.y;
```

```
 return(p1);
 }
```

0.3 Ulteriori considerazioni sulle strutture

Quando vengono passate come parametri, in C le strutture sono sempre passate per valore. Quindi, se dobbiamo modificare una struttura in una funzione, dobbiamo passare alla funzione un riferimento (puntatore) alla struttura.
Chiamiamo p il puntatore:



Per accedere ad esempio al campo x della struttura avremo due soluzioni equivalenti:

(*p).x

(è necessario porre il puntatore tra parentesi tonde, in quanto questo è un operatore con bassa precedenza);

p->x
(questa versione è preferibile).

Vediamo alcuni esempi di utilizzo. La seguente istruzione incrementa il campo x della struttura p di 10.

p->x = p->x+10;

Inoltre l'istruzione
++p->x
è equivalente a
++(p->x)
ed a
++((*p).x)
ed incrementa di 1 il campo x di p.

Scriviamo ora una funzione che, letti in input 2 numeri x ed y, sposta p di x ed y.

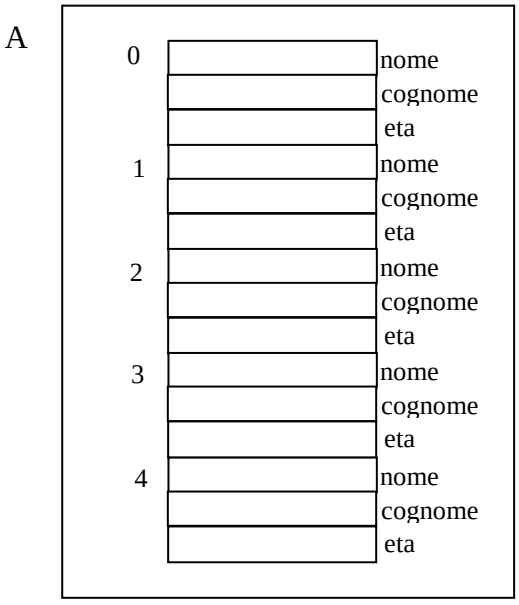
```
void shiftpoint (struct point *p) {
 int num;
 printf("\n ascissa spostata di   ");
 scanf("%d",&num);
 (*p).x+=num; /* è l'equivalente di (*p).x=(*p).x + num */
 printf("\n ordinata spostata di   ");
 scanf("%d",&num);
 (*p).y+=num; /* è l'equivalente di (*p).y=(*p).y + num */
 }
```

0.4 Array (vettori) di strutture

Esaminiamo ora una semplice implementazione delle strutture. Proviamo a realizzare una semplice rubrica che tiene conto di nome, cognome ed età di una persona (per semplicità, limitiamo la grandezza della rubrica a 5 elementi). Dato un array A[n], ogni elemento dell'array è una struttura definita come segue:

```
#define n 5;
struct person {
char nome[20];
char cognome[20];
int eta;
}A[n];
```

Quindi, ogni struttura è composta da due campi: un array di caratteri per il nome, uno per il cognome, ed un intero per l'eta'.



NOTA. Il C fornisce un operatore, l'operatore "typedef", che puo' essere utilizzato per semplificare la scrittura dei programmi, mediante la definizione di nuovi tipi, definiti dall'utente. Tramite la "typedef", e' possibile associare ad Una dichiarazione di tipo un nome, e poi riutilizzare successivamente tale nome per indicare il tipo associato.

Ad esempio, e' possibile riscrivere la dichiarazione delle struttura precedente come:

```
#define n 5;
typedef struct {
char nome[20];
char cognome[20];
int eta;
} person;
person A[n];
```

Si veda anche TESTO, Sezione 10.6.

0.4 Semplici operazioni con un array di strutture

0.4.1 Inserzione di elementi nell'array

Scriviamo la funzione "carica", la quale opera su un array A[n] di strutture di tipo "person", e carica in esso nomi, cognomi ed eta', letti da tastiera.

```
void carica (struct person A[], int n) {
int i; /*variabile locale, serve per scorrere l'array*/
for (i=0; i<n; i++) {
  printf("Nome della persona %d \n",i);
  scanf("%s",&A[i].nome); /*carico la stringa nel campo nome dell'i-esimo elemento del vettore*/
  printf("Cognome della persona %d \n",i);
  scanf("%s",&A[i].cognome); /*lo stesso per il cognome*/
  printf("Età della persona %d \n", i);
  scanf ("%d",&A[i].eta);
  }
}
```

0.4.2 Stampa dell'array

Vediamo ora la funzione “stampa” che scandisce l'array (avente n elementi) e stampa il contenuto di ogni singola struttura.

```
void stampa (struct person A[], int n){
  int i;
  for (i=0; i<n; i++){
    printf("Nome %s \n",A[i].nome);
    printf("Cognome %s \n", A[i].cognome);
    printf("Eta %d \n", A[i].eta);
  }
}
```

0.4.3 Costruzione di 2 array con condizione a partire da quello dato

Scriviamo la funzione “dividi” che genera un array C per i maggiorenni ed un array B per i minorenni, a partire da un array A con n elementi di tipo “person”. nB ed nC sono due parametri interi passati per riferimento, che conterranno, alla fine dell’esecuzione della funzione, il numero degli elementi (decrementato di uno) di B e di C rispettivamente.

```
void dividi(struct person A[], struct person B[], int *nB,
          struct person C[], int *nC)
{
  int i;
  *nB = 0; /* numero di elementi dell’array B dei minorenni */
  *nC = 0; /* numero di elementi dell’array C dei maggiorenni */
  for (i=0; i<n; i++)
    {
    if (A[i].eta < 18)
      {
      strcpy (B[*nB].nome,A[i].nome);
      strcpy (B[*nB].cognome,A[i].cognome);
      B[*nB].eta=A[i].eta;
      (*nB)++;
      }
    else
      {
      strcpy (C[*nC].nome,A[i].nome);
      strcpy (C[*nC].cognome,A[i].cognome);
      C[*nC].eta=A[i].eta;
      (*nC)++;
      }
    }
}
```

0.5 PUNTATORI IN C

Da integrare con TESTO, Sezioni 7.1, 7.2, 7.3
NOTA: anche se, come detto in TESTO, in C i puntatori possono assumere il valore '0' al posto di 'NULL', tale opzione NON e' una buona pratica di programmazione, e NON sara' considerata corretta nel corso.

Il tipo puntatore e' molto importante, in quanto, come si vedra' piu' vanti, supporta la gestione di **strutture dati dinamiche**. Inoltre, in C, i puntatori vengono esplicitamente utilizzati per trattare il passaggio dei **parametri per riferimento** (si riveda TESTO, Sezione 7.4). I puntatori, infatti, vengono utilizzati dai programmi per accedere alla memoria e per manipolare gli indirizzi.

Se *v* e' una variabile (di un qualunque tipo), ***&v*** e' la locazione od **indirizzo di memoria in cui e' memorizzato il valore di v**. In C i puntatori sono associati ad un TIPO di dato. Ad esempio, le dichiarazioni
int *p;
real *q;
dichiarano p e q di tipo puntatore ad intero e reale rispettivamente (e si tratta di due tipi *differenti*).

Indipendentemente dal tipo associato ad un puntatore, esiste tuttavia un unico valore, il valore **NULL**, per inizializzare variabili di tipo puntatore. Per utilizzare variabili di tipo puntatore nelle istruzioni dei programmi C, ci si avvale di due operatori:
- L'operatore **&** (operatore di indirizzo), che restituisce l'indirizzo (e non il valore!) del suo operando
- L'operatore ***** (operatore di indirezione o di dereferenziatione), che restituisce il valore dell'oggetto al quale punta il suo operando (di tipo puntatore).

Per meglio chiarire il significato di tali operatori, si consideri il seguente esempio
Si supponga di avere due variabili intere, a e b, inizializzate ad 1 e 2 rispettivamente.
(1) int a,b; a=1; b=2;
La dichiarazione
(2) int *p, *q;
dichiara p e q come variabili di tipo: puntatore ad intero. Gli assegnamenti:
(3) p=&a; q=&b;
hanno come effetto quello di associare a p l'indirizzo della (cella di memoria ch contiene la) variabile a, e similmente la seconda istruzione associa a q l'indizizzo di b. La situazione a questo punto puo' essere raffigurata come in figura Fig1.a.
Si noti che, per comodita' e chiarezza, il valore delle variabili di tipo puntatore (ovvero, il valore di p e di q) viene graficamente rappresentato come una freccia che punta ad una cella di memoria. In relata', tale valore e' proprio un indirizzo di memoria.
Nel seguito, si considerano alcuni possibili assegnamenti (non tutti corretti).

CASO 1 (Fig.1(b)) q=p;
Si tratta di un assegnamento fra variabili di tipo puntatore. Il valore di p (che e' un indirizzo di memoria) viene assegnato a q. Come risultato, p e q puntano alla stessa area di memoria. Si noti che se, successsivamente, il valore di a verra' modificato, tale modifica sara' "vista" sia da p che da q.

CASO 2 (Fig.1(c)) q=*p;
* e' l'operatore di deferenziazione. Il valore di *p e' il valore contenuto nella cella puntata da p (ovvero, il cui indirizzo costituisce il valore di p). In questo esempio, *p ha come valore il valore di a, ovvero un intero (il valore 1). Al contrario, q e' stato dichiarato come un puntatore (ovvero, un indirizzo). C'e' allora un **ERRORE** di tipi: non posso mettere un intero in una variabile che deve contenere puntatori (indirizzi).

CASO 3 (Fig.1(d)) *q=p;
p e' un puntatore, quindi il suo valore e' un indirizzo (l'indirizzo della cella a, nell'esempio). * e' l'operatore di deferenziazione. Quindi, assegnare un valore a *q significa assegnare un valore non alla cella q, ma alla cella indirizzata (puntata) da q, ovvero alla cella b. Tuttavia, b e' una cella preposta a contenere valori interi. C'e' allora un **ERRORE** di tipi: non posso mettere un puntatore in una variabile che deve contenere valori interi.

CASO 4 (Fig.1(e)) *q=*p;
In questo caso, l'operatore di deferenzianione * e' applicato sia a destra che a sinistra dell'assegnamento. A destra, *p ha come valore il contenuto della cella puntata da p (ovvero, l'intero 1). A sinistra, esso indica la cella puntata da q, ovvero la cella b, che e' preposta a contenere degli interi. Non c'e' quindi nessun errore di tipo: l'intero 1 viene assegnato alla cella 1 (si veda la figura).

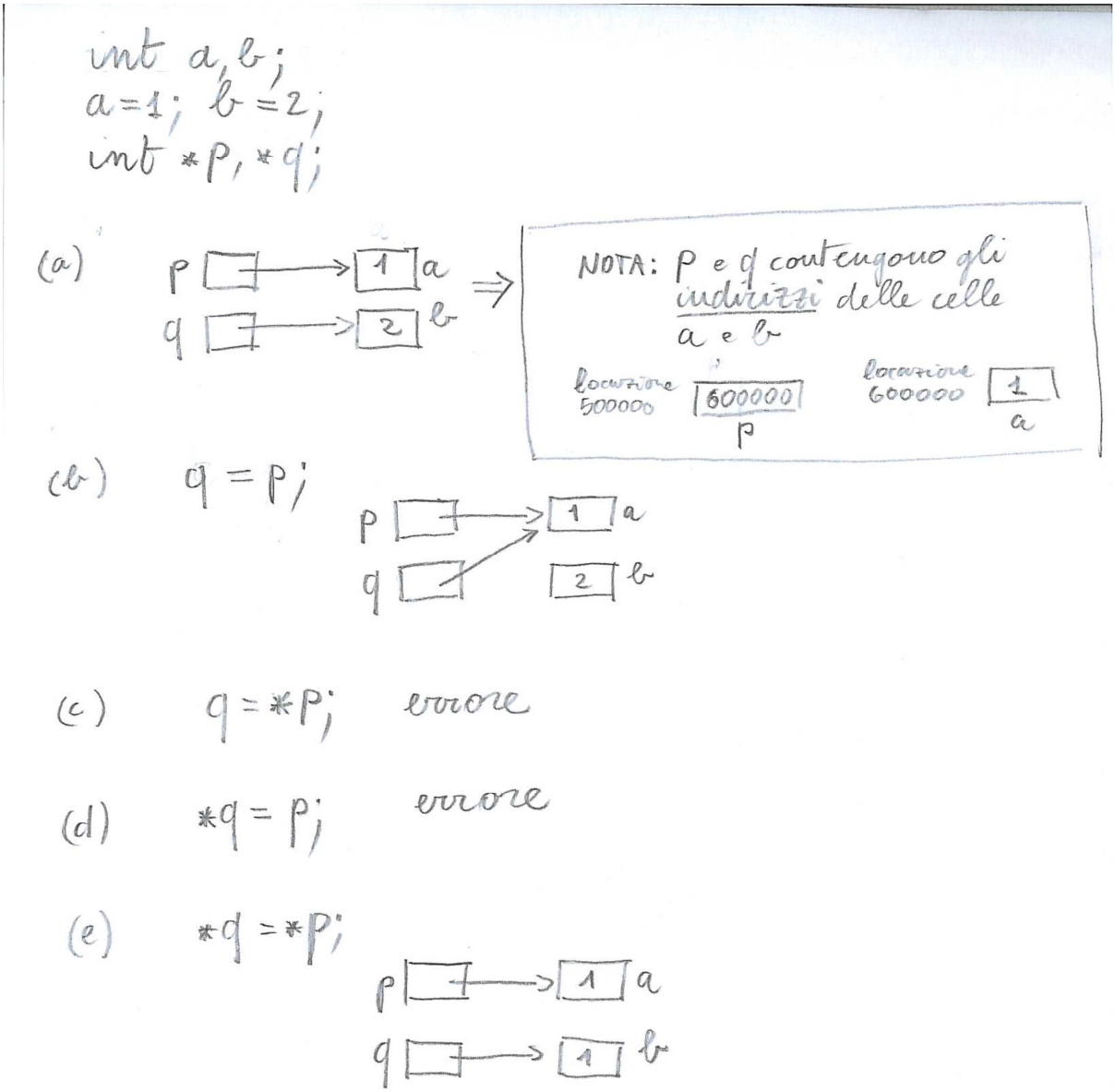

int a, b;
a=1; b=2;
int *p, *q;
(a)
p
q
1
a
2
b
NOTA: p e q contengono gli indirizzi delle celle a e b
locazione 500000
600000
p
locazione 600000
1
a
(b)
q = p;
p
q
1
a
2
b
(c)
q = *p; errore
(d)
*q = p; errore
(e)
*q = *p;
p
q
1
a
1
b

FIG. 1

0.6 ALLOCAZIONE DINAMICA DI MEMORIA: MALLOC e FREE

Si veda anche TESTO, Sezione 12.3.

Durante l'esecuzione di un programma C, lo spazio di memoria necessario all'esecuzione viene allocato per il programma, sotto forma di recod di attivazione. La dimensione di tali record di attivazione viene calcolata "staticamente", per ogni funzione, sulla base delle dichiarazioni delle funzioni stesse, Tuttavia, il C fornisce anche operatori per permettere, durante l'esecuzione dei programmi, di richiedere e\o liberare dinamicamente della memoria aggiuntiva (che viene presa dalla cosiddetta "Heap" di memoria). Si parla allora di "gestione dinamica" della memoria, dal momento che lo spazio di memoria allocata puo' variare dinamicamente durante l'esecuzione (di uno stesso record di attivazione).

L'operazione fornita dal C per richiedere dinamicamente della memoria e' l'operazione di "malloc". Supponiamo, ad esempio, di aver definito (mediante typedef) la struttura "person", come mostrato in Sezione 0.4 di queste Dispense.
L'istruzione

p = **malloc**(sizeof(person)

associa a p (che deve essere stata dichiarata di tipo "puntatore a person", mediante "person *p;") l'indirizzo di un nuovo spazio di memoria (preso dalla "Heap"), preposto a contenere una struttura di tipo person. L'istruzione di "**sizeof**" in essa contenuta calcola la quantita' di memoria necessaria a contenere il tipo che gli e' stato passato come parametro (**si veda TESTO: Sezione 7.7**).

Al contrario, l'istruzione

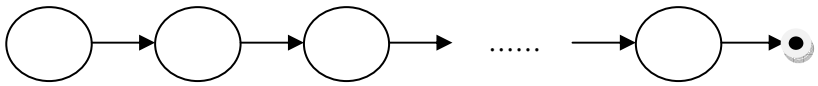
free(p)

"libera" lo spazio di memoria (nella "Heap") puntato dal puntatore p.

1 Funzioni iterative operanti su una lista

1.0 DEFINIZIONE DI LISTA E NOZIONI PRELIMINARI

Intuitivamente, una lista e' semplicemente una sequenza ordinata di elementi, solitamente detti "nodi". Graficamente. Essa puo' essere rappresentata come mostrato nel seguito, ove ogni ovale rappresenta astrattamente un elemento (nodo), e gli archi rappresentano la relazione di sequenza fra i nodi. Un nodo speciale viene utilizzato per rappresentare la fine della lista.



Piu' formalmente, e' possibile dare la seguente definizione astratta (e ricorsiva) per le liste:

DEFINIZIONE di LISTA.
(i) Una lista vuota e' una lista
(ii) Un nodo seguito da una lista e' una lista.

Questa definizione ricorsiva puo' apparire, ad una prima visione, astrusa. In realta' definisce chiaramente titte le possibili liste (di lunghezza arbitraria).

1. Il caso (i) (detto caso base) ci dice che la lista vuota e' una lista.
2. Combinando il caso (ii) con il caso (i), possiamo definire le liste con un solo elemento; il caso (ii) ci dice che un elemento seguito da una lista e' una lista. Ma, al passo 1, abbiamo appena dimostrato che la lista vuota e' una lista. Allora, qui dimostriamo che un elemento (seguito dalla lista vuota) e' una lista.
3. Applicando due volte il caso (ii) ed una il caso (i), definiamo le liste con due elementi; tramite il caso (i), abbiamo che un elemento seguito la una lista e'' una lista. Ma al passo 2, abbiamo dimostrato che la lista con un elemento e' una lista. Quindi, anche una sequenza di due elementi (seguiti dalla lista vuota) e' una lista.
4. Cosi' via, fino a definire liste di lunghezza arbitraria.

E' importante fare alcune osservazioni di carattere generale sulle liste.
(1) La lunghezza di una lista NON e' parte della definizione della lista stessa. Questo differenzia le liste, ad esempio, dagli array (vettori), e fa si' che vengano generalmente chiamate "strutture dati dinamiche". Una lista non ha una lunghezza prefissata, e puo' crescere o decrescere nel tempo (si pensi, ad esempio, alla lista ordinata alfabeticamente dei clienti di una banca).
(2) La lista vuota (ovvero, la lista che non contiene alcun nodo) e' una lista a tutti gli effetti (e non un errore!)
(3) E' parte della definizione di lista la nozione di sequenza ("seguito da"; nella definizione, ogni elemento indica il suo successivo)

1.0.1 IMPLEMENTAZIONE DI LISTE MEDIANTE PUNTATORI

Esistono differenti implementazioni alternative di LISTE in C. Nel resto del corso, ci focalizeremo esclusivamente su una di esse: quella dinamica, utilizzzando i puntatori.

Questa implementazione costituisce un modo molto "naturale" di gestire le liste, in quanto:
(1) I puntatori possono essere uutilizzati per implementare la relazione "seguito da", associando a ciascun nodo un puntatore al nodo successivo
(2) I puntatori permettono di gestire in modo efficiente il fatto che le liste sono strutture dati dinamiche, la cui lunghezza varia nel tempo. Infatti, tramite le operazioni di malloc e di free (applicabili solo a variabili di tipo puntatore), e' possibile ottenere nuova memoria (ad esempio per aggiungere un nuovo nodo ad una lista) oppure liberare memoria (ad esempio, per gestire la rimozione di un nodo da una lista).

In generale, la prima operazione da fare per definire una lista e' definire i tipi dei nodi della lista. Tipicamente, tali tipi possono essere gestiti in C tramite "struct" (strutture). Ad esempio, volendo gestire come lista dinamica gli impiegati di una ditta (ad esempio, in ordine alfabetico), potremmo utilizzare il tipo "persone" precedentemente descritto, e qui riportato per chiarezza espositiva.

```
typedef struct {
char nome[20];
char cognome[20];
int eta;
} person;
```

Poiche' pero' vogliamo definire una lista di persone, ogni nodo dovra' indicare chi e' il nodo che lo segue. Nella implementazione con i puntatori, questo puo' essere fatto aggiungendo un ulteriore campo alla struttura (tipicamente chiamato "next"), contenente un puntatore all'elemento successivo. Proseguendo l'esempio, potremmo quindi definire il tipo "person_node" come segue.

```
typedef struct person {
char nome[20];
char cognome[20];
int eta;
person *next;
} person_node;
```

Si noti che la definizione di person_node e' autoreferenziale, ed e' stato possibile ottenerlo solo menzionando un nome "provvisorio" per la struttura ("person" nell'esempio) prima della definizione della struttura stessa, per poi utilizzarlo come tipo del campo "next". Cio' e' possibile in C (e, eventualmente in forme diverse, nella maggior parte dei linguaggi di programmazione).

Per brevia', nel resto delle Dispense, verranno utilizzati dei nodi in cui, oltre al campo "next", esiste un unico altro campo contenente informazione, il campo DATA, di tipo intero. Le operazioni che proporremo sulle liste (inserzione di nodi, cancellazioni, ricerche) sono quasi totalmente indipendenti dal tipo di informazione associata ai nodi, e possono quindi essere facilmente applicati anche a nodi piu' ricchi, come "person_node" sopra descritto.

In particolare, nel resto delle Dispense, si considereranno liste definite come segue:

Le seguenti funzioni prevedono l'inclusione dei seguenti files di libreria:

```
#include <stdio.h>
#include <stdlib.h>
```

Le strutture dati utilizzate nel seguito per rappresentare gli elementi in una lista sono i seguenti:

```
typedef  int DATA;

struct linked_list
  {
        DATA   d;
        struct linked_list  *next;
  };

typedef struct linked_list ELEMENT;
typedef ELEMENT * LINK;
```

NOTA_1. Queste definizioni sono molto simili, ma NON IDENTICHE a quelle del testo (Sezione 12.4 del TESTO). Il TESTO usa le seguenti definizioni, dando nomi diversi ai vari tipi e campi (in particolare, il puntatore ad una lista viene chiamato "ListNodePtr" , mentre nelle dispense e' chaiamato "LINK") ed avendo come contenuto informativo del nodo (campo "data") un valore di tipo "char" (mentre nelle Dispense il campo si chiama "d" ed il valore e' di tipo intero):

```
struct listNode
  {
        char      data;
        struct listNode *nextPtr;
  };

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;
```

NOTA_2. Come gia' precedentremente menzionato, in generale la definizione del contenuto informativo dei nodi in una lista (ovvero, del tipo "DATA") risulta essere una struttura complessa (ad esempio, nome, cognome, eta', stipendio, indirizzo … di persone).

NOTA_3. Prima di iniziare, e' importante evidenziare come, date le definizioni di tipi prima mostrate (in particolare di LINK, che e' un puntatore ad un ELEMENT), una **lista** risulta univocamente individuata da una variabile di tipo LINK, ovvero da un **puntatore al primo elemento della lista**. Nel seguito, useremo spesso il termine "lista" per intendere "il puntatore al primo element\nodo della lista". Ad esempio, dicendo che una funzione ha in input (come parametro) una lista, intendiamo che ha come parametro un puntatore al pirimo elemento della lista (o NULL, se la lista e' vuota); dicendo che una funzione restituisce in output una lista, intendiamo che restituisce in output un puntatore al primo elemento della lista (o NULL, se la lista e' vuota). Queste considerazioni sono molto importanti considerando il passaggio delle liste come **parametro**. In particolare, passare una **lista come parametro per valore** ad una funzione f significa che, nel suo record di attivazione, la funzione f utilizzera' una copia del puntatore alla testa della lista (e non, assolutamente, una copia di tutti gli elementi della lista!). Analogamente, una **lista passata per riferimento** e' in realta' un puntatore al puntatore al primo elemento della lista.

1.0.2 ALLOCAZIONE MEMORIA

Al fine di allocare lo spazio (preso dall'heap di memoria) per un nuovo elemento di una lista, nel seguito verra' utilizzata la seguente funzione, che restituisce un puntatore al nodo allocato.

```
LINK newnode(void)
{
  return malloc(sizeof(ELEMENT));
  /* includere <stdlib.h> */
}
```

Per la disallocazione, viene utilizzata direttamente la funzione **free** (richiamata su di un puntatore ad ELEMENT, ovvero su di un LINK).

1.1 VISITA DI UNA LISTA

Tutte le funzioni di visita di una lista hanno una lista come parametro di input, passata **per valore**, dal momento che le visite non modificano la lista stessa. Come spiegato in nota, questo significa, in realta', che le funzioni sono richiamate su un puntatore al primo elemento della lista e, dal momento che viene utilizzato il passaggio per valore, la funzione utilizza al suo interno una **copia di tale puntatore**.

1.1.1 VISITA INCONDIZIONATA

Scansione di tutti gli elementi della lista.

/* Stampa degli elementi di una lista */
```
void printlis(LINK lis)
{
 while (lis != NULL)
     {
         printf(“%d\n", lis->d);
         lis= lis->next;
     }
}
```
Complessita': O(n) ove n e' il numero di nodi della lista

PATTERN GENERALE DI SOLUZIONE:
```
void funz(LINK lis)
{
 while (lis != NULL)
     {
         OPERAZIONE SUL SINGOLO ELEMENTO
         lis= lis->next;
     }
}
```

1.1.2 VISITA CON CONDIZIONE

Tutti gli elementi della lista devono essere visitati, ma le operazioni sono fatte solo su quegli elementi che soddisfano una certa condizione

/* Esempio: stampa i numeri della lista piu' grandi di x */

```
void printgreater(LINK lis, int x)
{
 while (lis != NULL)
    {
            if (lis->d > x)
            {printf("%d\n", lis->d);}
            lis= lis->next;
    }
}
```

Complessita': O(n) ove n e' il numero di nodi della lista

PATTERN GENERALE DI SOLUZIONE:

```
void funz(LINK lis, ....)
{
 while (lis != NULL)
    {
       OPERAZIONE CONDIZIONATA SUL SINGOLO ELEMENTO
       lis= lis->next;
    }
}
```

1.1.3 VISITA CON CONTATORE/ACCUMULATORE E CONDIZIONE

Tutti gli elementi della lista devono essere visitati, ma le operazioni sono fatte solo su quegli elementi che soddisfano una certa condizione. La condizione deve essere valuatata sulla base di un valore da aggiornare ad ogni passo della visita.

/* Esempio: stampa i numeri della lista in posizione multipla di x */

```
void printpos(LINK lis, int x)
{
int pos=1;
while (lis != NULL)
    {
            if ((pos % x)==0) {printf(">>>> %d\n", lis->d);}
            pos++;
            lis= lis->next;
    }
}
```

Complessita': O(n) ove n e' il numero di nodi della lista

PATTERN GENERALE DI SOLUZIONE

```
void funz(LINK lis, ....)
{
INIZIALIZZAZIONE VARIABILE
while (lis != NULL)
    {
            OPERAZIONE SUL SINGOLO ELEMENTO, CONDIZIONATA AL VALORE DELLA VARIABILE;
            AGGIORNAMENTO VALORE DELLA VARIABILE;
            lis= lis->next;
    }
}
```

1.1.4 VISITA DI PARTE DELLA LISTA

La lista deve essere visitata solo in parte. La parte da visitare dipende da un valore da aggiornare ad ogni passo della visita.

```
/* Esemio: stampa i primi n numeri della lista (se ci sono) */
void printn(LINK lis, int n)
{
int pos=1;
while ((lis != NULL) && (pos <= n))
    {
            printf("%d\n", lis->d);
           lis=lis->next;
            pos++;}
    }
}
```

Complessita': O(minimo(n,x)), ove n e' il numero di elementi nella lista

PATTERN GENERALE DI SOLUZIONE

```
void funz(LINK lis, ....)
{
INIZIALIZZAZIONE VARIABILE
while ((lis != NULL) && CONDIZIONE SULLA VARIABILE
    {
            OPERAZIONE SUL SINGOLO ELEMENTO;
            AGGIORNAMENTO VALORE DELLA VARIABILE;
            lis= lis->next;
    }
}
```

1.1.5 VISITA "CON FINESTRA"

La risoluzione di alcuni problemi richiede di considerare gli elementi di una lista non isolatamente, ma a 2 a 2 (o a 3 a 3, ecc.)

/* Esemio: restituisci in output il numero dei nodi della lista che sono somma dei loro due predecessori immediati */
ESEMPIO: LIS: 3 → 4 → 7 → 11 → 1 → 2 → 4 → 6 → 2
Output: 3

```
int printsum(LINK lis)
{
int count = 0;
if (lis == NULL) return 0;
else if (lis->next == NULL) return 0;
else
            while (lis->next->next != NULL)
             {
                        if (lis->next->next->d == lis->next->d + lis->d) count = count +1;
                        lis= lis->next;
            }
return count;
}
```

Complessita': O(n) ove n e' il numero di nodi della lista

1.1.6 COMPOSIZIONE DI “PATTERN”

Naturalmente, la soluzione di problemi leggermente piu’ complessi puo’ richiedere la cobinazione di due o piu’ dei pattern “elementari” di soluzione individuati.

NOTA. Questo aspetto e’ evidenziato solo a questo punto delle dispense, ma VALE PER TUTTI gli algoritmi e pattern di soluzione presentati anche piu’ avanti.

/* Esemio: stampa i numeri in posizione multipla di pos e non oltre l’ennesima posizione della lista*/

```
void print_pos_n(LINK lis, int pos,  int n)
{
int p=1;
while ((lis != NULL) && (p <= n))
    {
            if ((p%pos)==0) printf("%d\n", lis->d);
           lis=lis->next;
            p++;}
    }
}
```

Complessita’: O(minimo(n,x)), ove n e’ il numero di elementi nella lista

/* Esempio: sommatoria condizionata: somma solo i numeri piu' piccoli di x ed in posizione multipla di n */

```
int sumliscond(LINK lis, int x, int n)
{
  int acc=0;
  int pos=1;
  while (lis != NULL)
  {
            if (((pos % n)==0) && (lis->d < x))
            acc+=lis->d;
            lis=lis->next;
            pos++;
  }
  return(acc);
}
```

1.2 RICERCA DI UN ELEMENTO

Sono possibili differenti tipologie di ricerca

Ricerca di un elemento in input (prima occorrenza)
/* restituisce un puntatore alla prima occorrenza di x in una lista p. Se x non e' presente nella lista, restituisce NULL */

```
LINK find(int x, LINK p)
{
    int trovato=0;
    while ((p != NULL) && (! trovato))
       {
            if (p->d==x) trovato=1;
            else p=p->next;
            }
            return(p);
}
```

Ricerca di un elemento sulla base della posizione. In questo caso la "visita" deve prevedere un contatore di posizione.

/* restituisce un puntatore all'elemento n-esimo di una lista, se presente. Altrimenti, restituisce NULL */

```
LINK nth(int n, LINK p)
/* n => 1 */
{
int pos=1;
while ((pos<n)&&(p != NULL))
  {
            p=p->next;
            pos = pos+1;
  }
  return(p);
}
```

/* funzione che restituisce un puntatore alla n-esima occorrenza di x in lis */
/* versione che utilizza la find */

```
LINK findnth(int x, int n, LINK lis)
{
int count=0;
int trovato=0;
while ((lis != NULL) && (!trovato))
  {
            lis=find(x,lis);
            if (lis != NULL)
                        if (count+1 == n)
                                    trovato=1;
                        else
                                    {count=count+1; lis=lis->next;}
  }
return(lis);
}
```

/* funzione che restituisce un puntatore al nodo che precede la prima occorrenza di x nella lista lis –NULL se non c'e' x in lis;
Vedremo nel seguito che tale operazione di ricerca puo' essere utile al fine di determinare la posizione ove inserire/cancellare nodi */

```
LINK findpred(int x, LINK lis)
{
int trovato=0;
if (lis == NULL)
  {printf("lista vuota\n"); return(NULL);}
else
  {
            if (lis->d == x)
            {printf("%d e' a inizio lista\n", x); return(NULL);}
            else
            {
            while ((lis->next != NULL) && (! trovato))
                        if (lis->next->d == x)
                                    trovato=1;
                        else
                                    lis=lis->next;
            if (trovato)
                        return(lis);
            else
                        return(NULL);
    }
  }
}
```

1.3 CREAZIONE DI UNA LISTA

Creazione iterativa di una lista di numeri naturali maggiori di zero, leggendo l'input da tastiera.

```
/* Costruisce una lista di nodi con campo .d contenente interi positivi leggendo gli interi da tastiera. 0 indica la terminazione della lista */
LINK buildlis()
{
    int x;
    LINK lis, p, last;
    printf("nuovo numero da inserire in lista:\n");
    scanf("%d", &x);
    if (x<=0)
    lis= NULL; /* caso di lista vuota */
    else
       {
        /* inserzione del primo elemento in una lista */
          last=newnode();
          lis = last;
          last->d = x;
          last->next = NULL;
          printf("nuovo numero da inserire in lista:\n");
          scanf("%d", &x);
          while (x>0)
          {
               p=newnode();
               p->d = x;
               p->next = NULL;
               last->next = p;
               last = p;
               printf("nuovo numero da inserire in lista:\n");
               scanf("%d", &x);
          }
      }
    return(lis);
}
```

/* versione alternativa della funzione buildlis. Restituisce il risultato in un parametro passato per riferimento, anziche' direttamente come risultato della funzione. La complessita' e' ovviamente la stessa */

```
void buildlis2(LINK *lis)
{
    int x;
    LINK p, last;
    printf("nuovo numero da inserire in lista:\n");
    scanf("%d", &x);
    if (x<=0)
    *lis= NULL;
    else
       {
          last=newnode();
          *lis = last;
          last->d = x;
          last->next = NULL;
          printf("nuovo numero da inserire in lista:\n");
          scanf("%d", &x);
          while (x>0)
          {
              p=newnode();
              p->d = x;
              p->next = NULL;
              last->next = p;
              last = p;
              printf("nuovo numero da inserire in lista:\n");
              scanf("%d", &x);
          }
      }
}
```

1.4 MODIFICA DI UNA LISTA

1.4.1 MODIFICA DEL CONTENUTO DEI NODI

Se la modifica non riguarda la "struttura" della lista (i puntatori), ma solo il contenuto dei nodi, questa si reduce in pratica ad una semplice visita della lista stessa. La lista puo' essere passata per valore, e viene comunque modificata.

```
/* Esempio: aumenta di x il contenuto dei nodi in posizione multipla di n, solo se sono piu' grandi di y */
void addcond(LINK lis, int x, int y, int n)
{
int pos=1;
while (lis != NULL)
    {
        if (((pos % n) == 0) && (lis->d > y)) lis->d+=x;
        pos++;
        lis= lis->next;
    }
}
```

1.4.2 MODIFICA DI UNA LISTA: INSERZIONE DI NODI

Nel seguito verranno analizzati alcuni problemi riguardanti l'inserzione di uno o piu' nuovi nodi in una lista. Si noti che, qualora l'inserzione possa anche modificare la testa della lista, questa DEVE essere passata PER RIFERIMENTO alla funzione.

INSERZIONE IN TESTA

```
/* inserzione di un nuovo nodo contenente x in testa alla lista lis */
void headinsert(LINK *lis, int x)
{
    LINK p;
    p=newnode();
    p->d=x;
    p->next=*lis;
    *lis=p;
}
```

INSERZIONE IN CODA

```
/* inserzione di un nuovo nodo contenente x in coda alla lista */

void tailinsert(LINK *lis, int x)
{
  LINK p, q;
  q=*lis;
  p=newnode(); p->d=x; p->next=NULL;
  if (q == NULL)
  *lis=p;
  else
  {
        while (q->next != NULL) q=q->next;
        q->next=p;
  }
}
```

INSERZIONE ALL'INTERNO DI UNA LISTA

In questo caso, occorre abbinare il procedimento di inserzione ad uno di ricerca, atto a stabilire dove il nuovo nodo debba essere inserito. Tale ricerca DEVE trovare il nodo della lista da modificare che PRECEDE il nuovo nodo, ed agganciarlo

Nel seguito, si mostra invece un esempio completo, in cui la posizione dell'inserimento deve essere determinata mediante una visita con ricerca nella lista in input.

```
/* inserzione di newn prima della prima occorrenza di x in lis (se c'e')*/
void insertbefore(int x, int newn, LINK *lis)
{
int trovato=0;
LINK p, head;
head=*lis;
if (head == NULL)
  {printf("lista vuota\n");}
else
  {
  if (head->d == x)
    {
            p=newnode();
            p->d=newn;
            p->next=head;
            *lis=p;
    }
  else
    {
            while ((head->next != NULL) && (! trovato))
                    if (head->next->d == x)
                            trovato=1;
                    else
                            head=head->next;
            if (trovato)
                    {
                            p=newnode();
                            p->d=newn;
                            p->next=head->next;
                            head->next=p;
                    }
            else printf("element non trovato\n");
    }
  }
}
```

1.4.3 MODIFICA DI UNA LISTA: CANCELLAZIONE DI NODI

Nel seguito verranno analizzati alcuni problemi riguardanti la cancellazione di uno o piu' nuovi nodi in una lista. Si noti che, qualora la cancellazione possa anche modificare la testa della lista, la lista DEVE essere passata PER RIFERIMENTO alla funzione.

CANCELLAZIONE IN TESTA

```
/* cancella il primo nodo di una lista, liberando la memoria */
void disposefirst(LINK *lis)
{
LINK p;
    if (*lis != NULL)
    {
        p=*lis;
        *lis=(*lis)->next;
        free(p);
    }
}
```

CANCELLAZIONE IN CODA

```
/* cancella l'ultimo nodo di una lista, liberando la memoria
Si noti che occorre passare lis per riferimento, se no la modifica potrebbe andare persa (qualora la lista lis abbia un solo nodo)*/
void disposelast(LINK *lis)
{
LINK p;
    p=*lis;
    if (p != NULL)
    {
        if (p->next != NULL)
        {
           while (p->next->next != NULL) p=p->next;
           free(p->next);
           p->next=NULL;
        }
        else {p=*lis; *lis=NULL; free(p);}
    }
}
```

CANCELLAZIONE DI TUTTI GLI ELEMENTI

```
/* cancella tutti i nodi di una lista, liberando la memoria */
void disposelis(LINK *lis)
{
    LINK p;
    while (*lis != NULL)
    {
        p=*lis;
        *lis=(*lis)->next;
        free(p);
    }
}
```

CANCELLAZIONE "INTERNA" ALLA LISTA
In questo caso, occorre visitare la lista per determinare i nodi da cancellare, e procedure via via alla cancellazione

```
/* cancellare la prima occorrenza di x in una lista (utilizza la funzione di ricerca "findpred" definita prima) */
void delfirstx(int x, LINK *lis)
{
LINK p, q;
if (*lis != NULL)
   if ((*lis)->d==x) {p=*lis; *lis=(*lis)->next; free(p);}
   else
            {p=findpred(x,*lis);
            if (p != NULL)
                     {q=p->next;p->next=p->next->next; free(q);}
  }
}


/*cancellare tutte le occorrenze di x da una lista -- versione a 2 cicli*/
void delallx(int x, LINK *lis)
{
int uguali=1;
LINK p, head;
/* nel primo ciclo, estraggo solo le x che si trovano in testa alla lista */
while ((*lis != NULL)&&(uguali==1))
  {
   if ((*lis)->d==x)
            {
             p=*lis;
             *lis=(*lis)->next;
             free(p);
            }
   else uguali=0;
  }
  /* ora inizio le estrazioni non dalla testa */
if (*lis != NULL)
  {
            head=*lis;
            while (head->next != NULL)
            {
            if (head->next->d==x)
                     {
                     p=head->next;
                     head->next=head->next->next;
                     free(p);
                     }
            else head=head->next;
            }
  }
}
```

/*cancellare tutte le occorrenze di x da una lista -- versione a un ciclo*/

```
void delallxV2(int x, LINK *lis)
{
LINK p, head;
if (lis != NULL)
  {
    head=*lis;
    while (head->next != NULL)
            {
            if (head->next->d==x)
                    {
                    p=head->next;
                    head->next=head->next->next;
                    free(p);
                    }
            else head=head->next;
            }
    if ((*lis)->d==x) {p=*lis; *lis=(*lis)->next; free(p);}
  }
}
```

ESERCIZIO. Semplificare le funzioni **dellallx** e **delallxV2**, mediante l'uso della funzione **findpred** definita in precedenza**.**

2. FUNZIONI ITERATIVE OPERANTI SU 2 O PIU' LISTE

2.1 GENERAZIONE DI UNA NUOVA LISTA SULLA BASE DI UNA LISTA IN INPUT

Il caso piu' semplice di generazione di una nuova lista sulla base di una lista in input e' la COPIA. La funzione duplist, data una lista in input, ne costruisce una copia (in nuove aree di memoria), che viene restituita in output (ovvero, viene restituito in output un puntatore al primo elemento della copia costruita)

```
LINK duplist(LINK l)
{
  LINK  p, head, tail;
  head=NULL;
  while (l != NULL)
  {
            p=newnode();
            p->d = l->d;
            p->next = NULL;
            if (head == NULL)
                        {head=p; tail=p;}
            else  {tail->next=p; tail=p;}
            l=l->next;
  }
  return head;
}
```

La funzione, anche se semplice, mostra un PATTERN generale di costruzione iterativa di una lista, a partire da una lista in input.

1. **LINK p, head, tail**

Definizione di tre variabili locali di tipo puntatore. **p** serve per la generazione dei nuovi nodi; **head** e **tail** saranno rispettivamente la testa e la coda della nuova lista

2. **while (l != NULL)**
   { .....
   l = l-> next;
   }

Viene visitata (tutta, in questo caso) la lista in input

3. **p=newnode();**
   p->d = l->d;
   p->next = NULL;

Viene generato un nuovo nodo, che copia (nel campo 'd') il nodo corrente della lista in input. Il campo next di tale nodo e' inizializzato a NULL.

4. Il nodo p viene inserito nella nuova lista (di output):
   if (head == NULL)
   {head=p; tail=p;}

Se head e' NULL, si tratta del primo nodo, ed occorre inserirlo in testa altrimenti inserzione in coda

   else {tail->next=p; tail=p;}

Si noti come questo PATTERN generale possa essere applicator in tutta una classe di problem simili.

/* Esempio: duplicazione condizionata di lista (es: si duplichino (nell'ordine) tutti i nodi di una lista che sono maggiori di x */

```
LINK duplistcond(LINK l, int x)
{
  LINK p, head, tail;
  head=NULL;
  while (l != NULL)
  {
        if (l->d > x)
     {        p=newnode();
              p->d = l->d;
              p->next = NULL;
              if (head == NULL) {head=p; tail=p;}
              else {tail->next=p; tail=p;}
      }
    l = l->next;
  }
return head;
}
```

2.2 VISITA DI DUE O PIU' LISTE IN INPUT

La visita di due o piu' liste nasconde un problema, visto che la lunghezza delle liste non puo' essere nota a priori. Non e' quindi possible sapere quale lista e' piu' lunga, e quando una delle liste termina. Ne consegue che tali funzioni debbano contemplare piu' casi.
Ad esempio, una funzione f che visiti due liste L1 ed L2, deve considerare (ad ogni passo) 4 casi:
(1) Sia L1 che L2 sono diverse da NULL
(2) L1 e' NULL e L2 e' diversa da NULL
(3) L2 e' NULL e L1 e' diversa da NULL
(4) Sia L1 che L2 sono NULL

Ovviamente, il problema e' combinatorio, per cui, ad esempio, con tre liste in input, i casi possibili diventano 8.

Un secondo aspetto da considerare e' in quail casi l'analisi debba proseguire. Ad esempio, il seguente problema e' semplice, perche' richiede che le due liste vengano visitate "in parallelo", ma solo (nel caso peggiore) fino a quando entrambe sono non vuote. In questo caso, la complessita' in tempo e' lineare rispetto alla lunghezza della lista piu' corta.

```
/* date due liste, restituisce 1 se sono uguali, 0 altrimenti */
int equallis(LINK l1, LINK l2)
{
  int uguali=1;
  while ((l1!=NULL)&&(l2!=NULL) && (uguali==1))
  {
            if (l1->d != l2->d)
                        uguali=0;
            else
                        {l1=l1->next; l2=l2->next;}
  }
  if (uguali && (l1==NULL) && (l2==NULL))
            return(uguali);
  else      return(0);
}
```

Si noti che la variabile "uguali" e' inserita nella condizione del ciclo while al fine di sospendere subito la visita, in caso di disuguaglianza.
Complessita' O(min(lunghezza(l1),lunghezza(l2))
Nel caso peggiore, si arriva fino alla fine di una delle due liste per determinare la disuguaglianza, oppure le due liste sono uguali e sono da percorrere fino alla fine.

Al contrario, altri problemi richiedono di visitare "in parallel" entrambe le liste, fino alla fine della piu' lunga. In questo caso, la complessita' e' lineare rispetto alla lunghezza della lista piu' lunga, ed il PATTERN di soluzione consigliato prevede TRE cicli while. Il primo gestisce i casi in cui entrambe le liste sono non NULL, il secondo tratta le eventuali "eccedenze" di nodi nella prima lista, ed il terzo eventuali "eccedenze" della seconda lista. Si noti che
(i) almeno uno dei tre cicli non viene mai eseguito
(2) se almeno una delle due liste in input e' vuota, il primo ciclo non viene mai eseguito

/* date due liste di interi, contare quante volte i nodi della prima lista sono maggiori dei nodi della seconda lista in posizione corrispondente. Qualora una delle due liste sia piu' corta dell'altra, si assuma che i nodi "mancanti" contengano il valore zero
Esempio
L1: 5 → -4 → 5
L2: 4 → 4 → 11 → -3 → 5
Output: 2 */

```
int countmag(LINK l1, LINK l2)
{
  int count=0;
  while ((l1!=NULL)&&(l2!=NULL))
  {
            if (l1->d > l2->d) count = count +1;
            l1 = l1->next;
            l2=l2->next;
}
            while (l1!=NULL)
{
            if (l1->d > 0) count = count +1;
            l1 = l1->next;
}
while (l2!=NULL)
{
            if (l2->d < 0) count = count +1;
            l2 = l2->next;
}
return(count);
}
```

Complessita': O(max(lunghezza(l1), lunghezza(l2))

Cio' corrisponde al seguente PATTER generale:
PATTERN per il percorrimento "in parallelo" di due liste l1 ed l2, fino alla terminazione della piu' lunga.

```
... f(LINK l1, LINK l2, ...)
{
  while ((l1!=NULL)&&(l2!=NULL))
  {
            OPERAZIONI;
            l1 = l1->next; l2=l2->next;
}
            while (l1!=NULL)
{
            OPERAZIONI;
            l1 = l1->next;
}
while (l2!=NULL)
{
            OPERAZIONI;
            l2 = l2->next;
}
eventuale return
}
```

Complessita': O(max(lunghezza(l1), lunghezza(l2))

Il seguente esercizio esemplifica la classe di problemi in cui ci sono tre liste in input, che devono essere percorse “in parallelo” fino al termine della piu’ lunga. Come motivato sopra, i casi da considerare sono 8. In alcuni problemi, e’ possible scrivere codice piu’ compatto mettendo insieme, se ce ne sono, i casi che si comportano in modo uguale. Dal punto di vista dell’analisi e’ pero’ FONDAMENTALE ricordardsi che tutti gli 8 casi devono essere considerati, e gestiti dalla funzione.

/* funzione che, date tre liste l1, l2 ed l3 in input, restituisce 1 se, posizione per posizione, il nodo di l3 contiene la somma del nodo di l1 e di l2. Le liste devono essere considerate come “riempite di zeri” fino a raggiungere tutte e tre la lunghezza della lista piu’ lunga.
Es.
l1: 4 → 2 → 5
l2: 3 → 3 → 2 → 5 → 6 → 0
l3: 7 → 5 → 7 → 5 → 6
output: 1 */

```
int issum?(LINK l1, LINK l2, LINK l3)
{
int valid=1;
while((l1!=NULL)&&(l2!=NULL)&&(l3!=NULL)&&(valid==1))
{           if (l3->d != l1->d+l2->d) valid=0;
            else {l1=l1->next; l2=l2->next; l3=l3->next;}
}
while((l1!=NULL)&&(l2!=NULL)&&(l3==NULL)&&(valid==1))
/* si noti che la condizione sopra puo’ essere semplificata con l’eliminazione di “(l3==NULL)”
Per chiarezza espositiva, tutte le condizioni sono qui espresse per esteso */
{           if (0 != l1->d+l2->d) valid=0;
            else {l1=l1->next; l2=l2->next; }
}
while((l1!=NULL)&&(l2==NULL)&&(l3!=NULL)&&(valid==1))
{           if (l3->d != l1->d) valid=0;
            else {l1=l1->next; l3=l3->next;}
}
while((l1==NULL)&&(l2!=NULL)&&(l3!=NULL)&&(valid==1))
{           if (l3->d != l2->d) valid=0;
            else {l2=l2->next; l3=l3->next;}
}
while((l1!=NULL)&&(l2==NULL)&&(l3==NULL)&&(valid==1))
{           if (l1->d!=0) valid=0;
            else {l1=l1->next;}
}
while((l1==NULL)&&(l2!=NULL)&&(l3==NULL)&&(valid==1))
{           if (l2->d!=0) valid=0;
            else {l2=l2->next;}
}
while((l1==NULL)&&(l2==NULL)&&(l3!=NULL)&&(valid==1))
{           if (l3->d!=0) valid=0;
            else {l3=l3->next;}
}
/* l’ottavo caso, ovvero il caso in cui l1, l2 ed l3 sono tutti e tre NULL, viene omesso,
perche’ in tale situazione non c’e’ piu’ alcuna oprazione da fare, se non la return di valid */

return(valid)
}
```

Complessita’: O(max(lungjezza(l1), lunghezza(l2), lunghezza(l3))

2.3 FUNZIONI SU PIU' LISTE (IN INPUT e OUTPUT)

Ovviamente, in casi piu' complessi, si possono avere piu' liste sia in input che in output ad una funzione.

Mostriamo un esempio di funzione che, date due liste in input da visitare "in parallelo", e fino alla fine della lista piu' corta, costruiscono una lista in output. La complessita' e', ovviamente, lineare rispetto alla lunghezza della lista piu' corta.

/* Date due liste l1 ed l2, costruire la lista "somma posizionale" di l1 ed l2, fermandosi non appena la piu' corta delle due liste termina
ad esempio:
l1: 2->4->6->8
l2: 5->4->3
risultato: 7->8->9 */

```
LINK buildsumlis_short(LINK l1, LINK l2)
{
  LINK p,  head,  tail;
  head=NULL; tail=NULL;
  while ((l1 != NULL) && (l2 != NULL))
  {
            p=newnode();
            p->d = l1->d+l2->d;
            p->next = NULL;
                if (head == NULL){head=p; tail=p;}
                    else {tail->next=p; tail=p;}
            l1 = l1->next;
            l2 = l2->next;
  }
return head;}
}
```

Complessita': O(min(lunghezza(l1), lunghezza(l2))

In generale, il PATTERN per la costruzione di una nuova lista ottenuta visitando "in parallelo" (posizione per posizione) due liste in input, fino alla fine della piu' corta, e' mostrato nel seguito.

```
LINK buildsumlis_short(LINK l1, LINK l2, ....)
{
  LINK p,  head,  tail;
  head=NULL; tail=NULL; /* inizializzazione lista di output */
  while ((l1 != NULL) && (l2 != NULL))
  {
            p=newnode();  /* creazione nuovo nodo */
            DETERMINAZIONE DI p->d DATI  l1->d ed l2->d;
            p->next = NULL;
                if (head == NULL){head=p; tail=p;} /* inserzione di p in testa */
                    else {tail->next=p; tail=p;} /* inserzione di p in coda */
            l1 = l1->next; l2 = l2->next; /* spostamento al nodo successivo, su entrambe le liste */
  }
return head;  /* return della nuova lista */
}
```

Complessita': O(min(lunghezza(l1), lunghezza(l2))

Mostriamo ora un esempio di funzione che, date due liste in input da visitare “in parallelo”, e fino alla fine della lista piu’ lunga, costruisce una lista in output. In questo caso, la complessita’ e’ lineare sulla lunghezza della lista piu’ lunga.

/* date due liste, costruire la lista "somma", posizione per posizione (considerando il valore zero per i nodi “mancanti” nella lista piu’ corta). Ad esempio:
l1: 2->4->6->8->4
l2: 5->4->3
risultato: 7->8->9->8->4 */

```
LINK build(LINK l1, LINK l2)
{
  LINK p,  head,  tail;
  head=NULL; tail=NULL;
  while ((l1 != NULL) && (l2 != NULL))
  {
            p=newnode();
            p->d = l1->d+l2->d;
            p->next = NULL;
                if (head == NULL){head=p; tail=p;}
                    else {tail->next=p; tail=p;}
            l1 = l1->next;
            l2 = l2->next;
  }
  while (l1 != NULL)
 {
            p=newnode();
            p->d = l1->d;
            p->next = NULL;
            if (head == NULL){head=p; tail=p;}
                    else {tail->next=p; tail=p;}
            l1 = l1->next;
  }
  while (l2 != NULL)
 {
            p=newnode();
            p->d = l2->d;
            p->next = NULL;
            if (head == NULL){head=p; tail=p;}
                    else {tail->next=p; tail=p;}
            l2 = l2->next;
  }
 return (head);
}
```

Complessita’: O(max(lunghezza(l1), lunghezza(l2))

Si noti che, volendo riutilizzare le funzioni gia’ definite, il secondo ed il terzo ciclo “while” in build possono essere sostituiti da opportune richiami della funzione duplist.

In generale, il PATTERN di una funzione che, date due liste in input da visitare “in parallelo”, e fino alla fine della lista piu’ lunga, costruisce una lista in output e’ mostrato nel seguito. Si notiono le analogie con il pattern della visita “in parallelo” di due liste in input! (questo pattern e’ in realta’ una *estensione* del precedente, per costruire anche una lista di output).

```
LINK f(LINK l1, LINK l2,...)
{
  LINK p,  head,  tail;
  head=NULL; tail=NULL; /* inizializzazione lista di output */
  while ((l1 != NULL) && (l2 != NULL))
  {
            p=newnode(); /* creazione nuovo nodo */
            DETERMINAZIONE DI p->d DATI  l1->d ed l2->d;
            p->next = NULL;
             if (head == NULL){head=p; tail=p;} /* inserzione di p in testa */
               else {tail->next=p; tail=p;} /* inserzione di p in coda */
            l1 = l1->next; l2 = l2->next; /* spostamento al nodo successivo, su entrambe le liste */
  }
  while (l1 != NULL)
 {
            p=newnode();
            DETERMINAZIONE DI p->d DATO  l1->d;
            p->next = NULL;
            if (head == NULL){head=p; tail=p;} /* inserzione di p in testa */
              else {tail->next=p; tail=p;} /* inserzione di p in coda */
            l1 = l1->next; /* spostamento al nodo successivo, sulla lista l1 */

  }
  while (l2 != NULL)
 {
            p=newnode();
            DETERMINAZIONE DI p->d DATO  l2->d;
            p->next = NULL;
            if (head == NULL){head=p; tail=p;} /* inserzione di p in testa */
              else {tail->next=p; tail=p;} /* inserzione di p in coda */
            l2 = l2->next; /* spostamento al nodo successivo, sulla lista l2 */

  }
 return (head);
}
Complessita’: O(max(lunghezza(l1), lunghezza(l2))
```

E’ evidente come ci siano parti di codice che si ripetono con regolarita’ nella soluzione di queste tipologie di esercizi. E’ quindi possible, al fine di rendere il codice piu’ compatto, definire ed utilizzare una funzione ausiliaria. La funzione create&insert ha in input un valore (val), la testa (head) e la coda (tail) di una lista, crea un nuovo nodo con campo ‘d’ uguale a ‘val’, e lo inserisce nella lista. Si noti che ‘head’ e ‘tail’ devono essere passate per riferimento.

```
void create&insert(int val, LINK *head, LINK *tail)
{
LINK p;
p=newnode();
p->d = val;
p->next = NULL;
if (*head == NULL){*head=p; *tail=p;} /* inserzione di p in testa */
            else {(*tail)->next=p; *tail=p;} /* inserzione di p in coda */
}
```

Utilizzando tale funzione, il PATTERN mostrato sopra puo' essere esemplificato come segue:

```
LINK f(LINK l1, LINK l2,...)
{
  LINK p,  head,  tail;
  int newval;
  head=NULL; tail=NULL; /* inizializzazione lista di output */
  while ((l1 != NULL) && (l2 != NULL))
  {
            DETERMINAZIONE DI newval DATI  l1->d ed l2->d;
            create&insert(newval, &head, &tail);
            l1 = l1->next; l2 = l2->next; /* spostamento al nodo successivo, su entrambe le liste */
  }
  while (l1 != NULL)
 {
            DETERMINAZIONE DI newval DATO  l1->d;
            create&insert(newval, &head, &tail);
            l1 = l1->next; /* spostamento al nodo successivo, sulla lista l1 */
  }
  while (l2 != NULL)
 {
            p=newnode();
            DETERMINAZIONE DI newval DATO  l2->d;
            create&insert(newval, &head, &tail);
            l2 = l2->next; /* spostamento al nodo successivo, sulla lista l2 */
  }
 return (head);
}
```

Complessita': O(max(lunghezza(l1), lunghezza(l2))

NOTA. La funzione create&insert non verra' piu' utilizzata negli esercizi nel seguito, in modo da rendere completamente ESPLICITE le funzioni definite (ovvero per chiarezza ed esaustivita' espositiva)

Come ulteriore esercizio viene mostrato un esempio complesso di concatenamento condizionato di due liste.

/* Costruire una nuova lista ottenuta concatenando gli elementi da posizione 1 a posizione n dei nodi di l1 con quelli da posizione m in poi di l2. Si noti che non puo' essere fatta alcuna ipotesi sulla lunghezza di l1 ed l2
Esempio:
l1: 3→5
l2:1→9→7→8→9→6→4→3
n= 4 m=6
output: 3→5→6→4→3
*/

In questo caso **non e' assolutamente necessario percorrere "in parallelo" le due liste** l1 ed l2. Si puo' percorrere e duplicare fino a posizione n (se l1 ha almeno n nodi) la lista l1, e poi agganciare alla nuova lista il risultato ottenuto percorrendo l2 fino a posizione m (ammesso che l2 abbia almeno m nodi), e poi duplicando tutti i seguenti**. Si lascia questa soluzione come esercizio per gli studenti.**
Tuttavia, nel seguito viene mostrato come, data la sua generalita', il PATTER di visita di due liste con costruzione di lista di output possa essere applicator per trattare anche questo esercizio. Si noti che, in questo caso, non si deve proseguire fino alla fine della lista piu' lunga, ma fino alla fine di l2 (ammesso che questa abbia almeno m nodi).

```
LINK concatenate(int n, int m, LINK l1, LINK l2)
{
  LINK p,  head,  tail;
  head=NULL; tail=NULL;
  int pos=1;
  while ((l1 != NULL) && (l2 != NULL))
  {
            if (pos <= n)
                      { p=newnode(); p->d = l1->d; p->next = NULL;
                       if (head == NULL){head=p; tail=p;}
                      else {tail->next=p; tail=p;} }
            if(pos >= m)
                      { p=newnode(); p->d = l2->d; p->next = NULL;
                       if (head == NULL){head=p; tail=p;}
                      else {tail->next=p; tail=p;} }
            l1 = l1->next;
            l2 = l2->next;
            pos++;
  }
  while (l1 != NULL)
 {
            if (pos <= n)
                      { p=newnode(); p->d = l1->d; p->next = NULL;
                       if (head == NULL){head=p; tail=p;}
                      else {tail->next=p; tail=p;} }
            l1 = l1->next; pos++;
  }
  while (l2 != NULL)
 {
            if (pos >=m)
                      { p=newnode(); p->d = l2->d; p->next = NULL;
                       if (head == NULL){head=p; tail=p;}
                      else {tail->next=p; tail=p;} }
            l2 = l2->next; pos++;
  }
 return (head);
}
```

Naturalmente, non in tutti i problemi le due (o piu') liste in input devono essere visitate "in parallelo". Ad esempio, si consideri la seguente funzione, che, date due liste ordinate, ne genera (con creazione di nuovi nodi) l'intersezione. Si noti che, poche' l'intersezione con una lista vuota e' vuota, la funzione deve terminare quando la lista piu' corta termina (ovvero, e' lineare sulla lunghezza della lista piu' corta). Qui l'idea e' che, dal momento che le due liste sono, per ipotesi, ordinate, ad ogni passo dell'iterazione e' possible confrontare un elemento della prima lista con uno della seconda. Se sono uguali, questo elemento viene duplicato ed inserito nella nuova lista, altrimenti si continua, avanzando SOLO nella lista che contiene l'elemento minore.

```
/* date due liste ordinate, generare la lista intersezione
Esempio:
L1: 2->4->6->8->10
L2: 3->4->6->7->9->10->13
Risultato: 4->6->10*/
LINK inters(LINK l1, LINK l2)
{
LINK p,head,tail;
head=NULL; tail=NULL;
while ((l1 !=NULL) && (l2 != NULL))
  {
   if (l1->d == l2->d)
              {
               p=newnode();
               p->d=l1->d;
               p->next=NULL;
               if (head==NULL){head=p; tail=p;}
               else      {tail->next=p; tail=p;}
               l1=l1->next;
               l2=l2->next;
              }
   else if (l1->d < l2->d) l1=l1->next;
   else l2=l2->next;
  }
return head;
}
```

Ovviamente, la costruzione di liste non necessariamente deve avvenire con la creazione di nuovi nodi. Viene ad esempio mostrato un esercizio in cui i nodi di una lista in input vengono ri-arrangiati in due liste di output.

/* data una lista lis in input, la decompone in due liste par e dis, contenente rispettivamente i nodi pari ed i nodi dispari di lis, nell'ordine in cui questi occorrono in lis. Non vengono creati nuovi nodi */

```
void splitpardis(LINK *lis, LINK *par, LINK *dis)
{
LINK lpar, ldis, q;
*par=NULL;
*dis=NULL;
while (*lis != NULL)
  {
  if (((*lis)->d % 2) == 0)
            {if (*par == NULL)
                        {*par=*lis; lpar=*lis;}
            else      {lpar->next=*lis; lpar=*lis;}}
  else
            {if (*dis == NULL)
                        {*dis=*lis; ldis=*lis;}
            else      {ldis->next=*lis; ldis=*lis;}}
  q=*lis;
  *lis=(*lis)->next;
  q->next=NULL;
  }
}
```

CAPITOLO 3. INTRODUZIONE ALLA RICORSIONE

Nel seguito sono proposte note integrative riguardanti le funzioni ricorsive in generale, e le funzioni RICORSIVE in C nello specifico. Particolare attenzione verra' dedicata allo sviluppo di funzioni ricorsive in C per trattare diverse tipologie di problemi su liste, di complessita' crescente.

3. INTRODUZIONE ALLA RICORSIONE

3.1 ESEMPIO E MODELLO DEI RECORD DI ATTIVAZIONE

La funzione fattoriale costituisce il classico esempio di funzione ricorsiva (si veda anche TESTO, Sezione 5.14)

Sui testi di matematica, il fattoriale e' spesso indicato con il simbolo "!", e definito come segue:

(i) 0! = 1
(ii) n! =n * (n-1)!

Per maggiore leggibilita' e per omogeneita' al modo "informatico" di denotare le funzioni, questa definizione puo' essere equivalentemente riscritta come segue

(i) fact(0) = 1
(ii) fact(n) = n * fact(n-1)

Tali definizioni sono ricorsive, perche' adottano una funzione nella definizione della funzione stessa. Tuttavia, esse sono perfettamente corrette, e permettono di definire, se pur "implicitamente", tutti i possibili valori assunti dalla funzione.

(1) fact(0) = 1 per definizione, parte (i)
(2) fact(1) = 1 * fact(0) per definizione, parte (ii). Ma, dato (1), fact(0)=1, e quindi, sostituendo, fact(1)=1*1=1
(3) fact(2) = 2 * fact(1) per definizione, parte (ii). Ma, dato (2), fact(1)=1, e quindi, sostituendo, fact(2)=2*1=2
(4) fact(3) = 3 * fact(2) per definizione, parte (ii). Ma, dato (3), fact(2)=2, e quindi, sostituendo, fact(3)=3*2=6
(5) .....

Si notino le analogie fra definizioni ricorsive come quella del fattoriale e la nozione matematica di **INDUZIONE**. Informalmente parlando, volendo ad esempio dimostrare per induzione che una proprieta' P vale su un insieme di elementi, si dimostra che tale proprieta' vale per un elemento (**passo BASE** della dimostrazione), e poi, **assumendo** che P valga su n elementi (**ipotesi induttiva**), si dimostra che P vale anche su n+1 elementi (**passo di induzione**). Ad esempio, per dimostrare che P vale sui numeri interi, si dimostra che

(1) P vale su 0 (**passo base**)
(2) Se P vale (**ipotesi induttiva**) su un generico numero naturale n, allora (**passo di induzione**) vale su n+1

In C, come in (quasi) tutti i linguaggi di programmazione, e' possible dare una definizione ricorsiva delle funzioni. Ad esempio, in C, la funzione fattoriale puo' essere correttamente definite come segue:

```
/* calcolo ricorsivo del fattoriale */
int fact(int n)
{
  if (n==0)
        return 1;
  else
        return n*fact(n-1);
}

/* passo base: fact(0)=1 */
/* ipotesi induttiva: fact(n-1) restituisce (n-1)! */
/* passo induttivo: n! = n*(n-1)! */
/* terminazione: ad ogni richiamo, n e' decrementata di uno, fino a raggiungere il valore 0 del passo base */
/* la complessita' delle funzioni ricorsive verra' discussa nelle NOTE: paragrafo 2.2)
      tempo: O(n)
      spazio: O(n) */
```

Naturalmente, la funzione fattoriale puo' essere definite in C anche in modo iterativo

```
int fact_iter(int n)
{
  int product = 1;
  while(n>1)
  {
            product=product*n;
            n=n-1;
  }
return(product);
}
```

Si noti tuttavia lo stile completamente differente di soluzione del problema. Nel caso della soluzione ricorsiva, il problema e' stato definito "in astratto" per induzione, descrivendo il passo base ed il passo induttivo. Al contrario, nella definizione iterativa, sono state esplicitate le operazioni (un ciclo di prodotti e sottrazioni, finche' il parametro n non diventa 1) da eseguire per raggiungere il risultato.

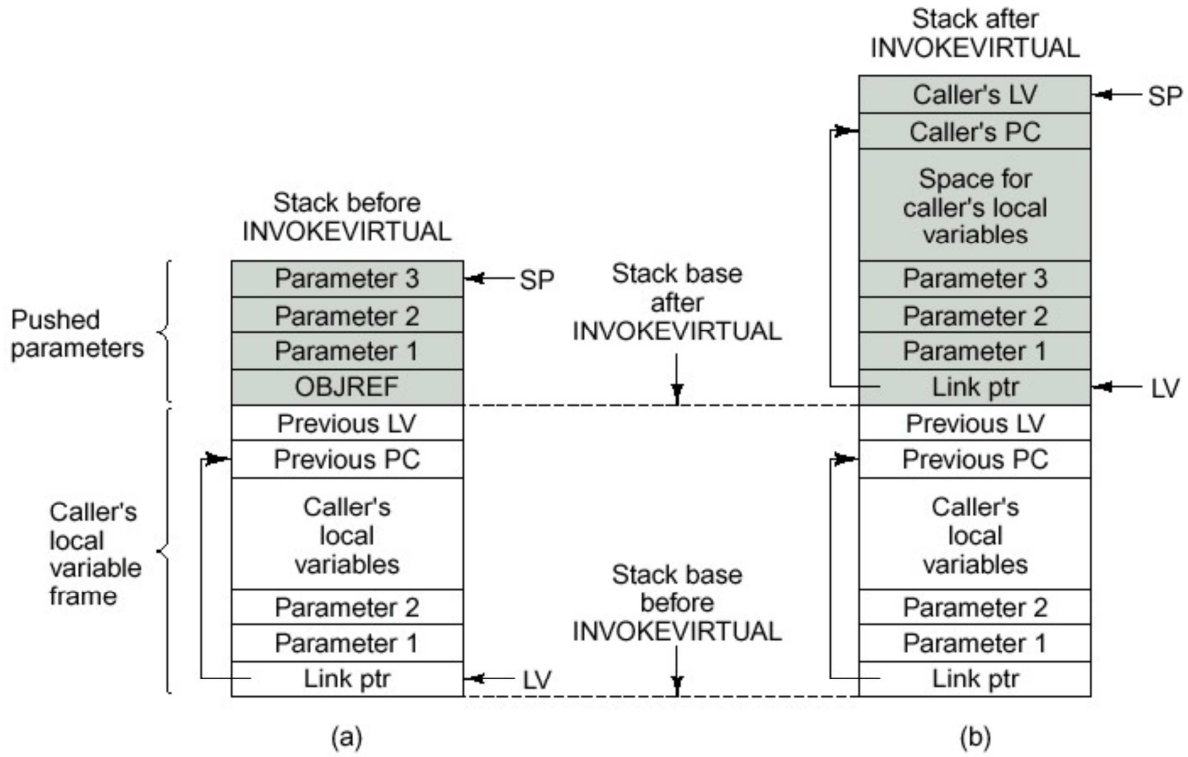
NOTA. Questa differenza puo' apparire poco significativa su problem semplici quail la definizione del fattoriale, ma puo' essere invece cruciale per problem piu' complessi, quali ad esempio "le Torri di Hanoi" (si vedano queste dispense, Sezione 3.4), o le funzioni di ordinamento (es. Quicksort, Mergesort; Sezione 6 di queste dispense), o ancora problemi concernenti strutture dati complesse quail alberi e grafi.

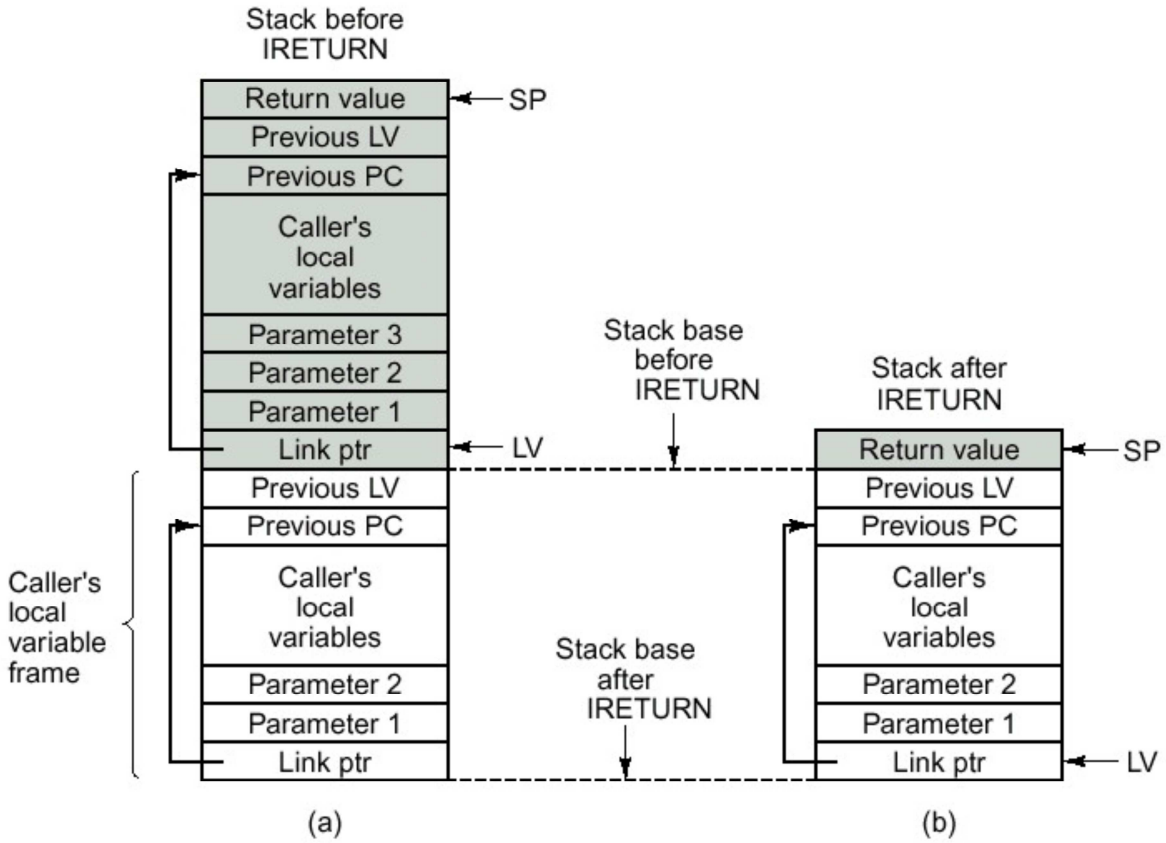
Il fatto che, tramite una definizione ricorsiva di funzione, non sia necessario esplicitare tutti i passi di computazione necessari alla risoluzione del problema (dal momento che ci si puo' avvalere, all'atto della definizione, dell'ipotesi induttiva), non significa che, a livello di ESECUZIONE di una funzione ricorsiva da parte del computer, tali passi non vengano eseguiti. Per andare ad analizzare piu' in dettaglio come avvenga l'esecuzione di una funzione ricorsiva possiamo avvalerci del modello dei RECORD DI ATTIVAZIONE.

Per tale modello, lo studente si puo' riferire al libro di testo del corso di Architetture (Tanenbaum).

TERMINOLOGIA: la nozione di "RECORD DI ATTIVAZIONE" corrisponde a quella di "FRAME DELLE VARIABILI LOCALI" adottata nel crso di Architetture 1.

Vengono qui riportate, per chiarezza, due figure tratte dal Tanenbaum, che illustrano la situazione della PILA (STACK) prima all'atto della chiamata di una funzione (allocazione di un nuovo record di attivazione sulla pila) e della fine di una chiamata (disallocazione di un record di attivazione dalla PILA).

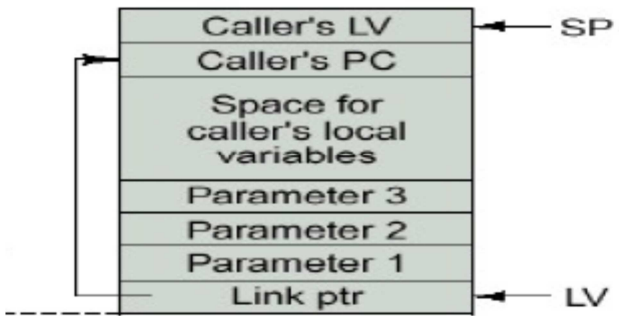




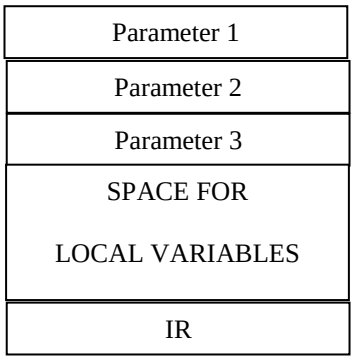
Tuttavia, al fine del presente corso, e' possible astrarre da alcuni dettagli di tale modello.

In particolare, il modello del record di attivazione viene semplificato come segue, astraendo da alcuni dettagli:

Modello nel Tanenbaum (Architetture 1)



Nuovo modello, utilizzato nel corso:



Si astrae quindi da:
LV = registro che punta alla base del frame delle variabili locali per la procedura in esecuzione
SP = registro che punta al top dello stack degli operandi
e si indica con IR (indirizzo di ritorno) il valore del PC (program counter) della funzione chiamante.

Inoltre, nel corso di Programmazione II, si adotta la convenzione di far crescere la PILA (STACK) dei record di attivazione dall'alto verso il basso.

Nel seguito, vedremo alcuni esempi di come le funzioni ricorsive sono eseguite, utilizzando il modello dei recod di attivazione SEMPLIFICATO. Per brevita', il termine "record di attivazione" verra' nel seguito abbreviato con "RA".

Esaminiamo dapprima l'esecuzione con i RA della versione ricorsiva *non di coda* (si veda la sezione 3.3 di queste note) della funzione fattoriale, riportata nel seguito per chiarezza.

```
int fact(int n)
{
  if (n==0)return 1;
  else return n*fact(n-1);
}
```

Supporremo nel seguito che tale funzione sia stata richiamata nel main di un programma con l'istruzione:
x = fact(3);
ed indicheremo con (i2) l'indirizzo nel main di questa istruzione di assegnamento. Indicheremo inoltre con (i1) l'indirizzo del prodotto n*fact(n-1), che e' l'indirizzo di ritorno dopo le chiamate ricorsive annidate. Infatti, l'esecuzione dell' l'istruzione
return n*fact(n-1)
comporta alcuni passi:
(i) valutazione (con una chiamata ricorsiva di fact(n-1))

(ii) esecuzione del prodotto n* risultato della chiamata ricorsiva
(iii) restituzione del valore della funzione

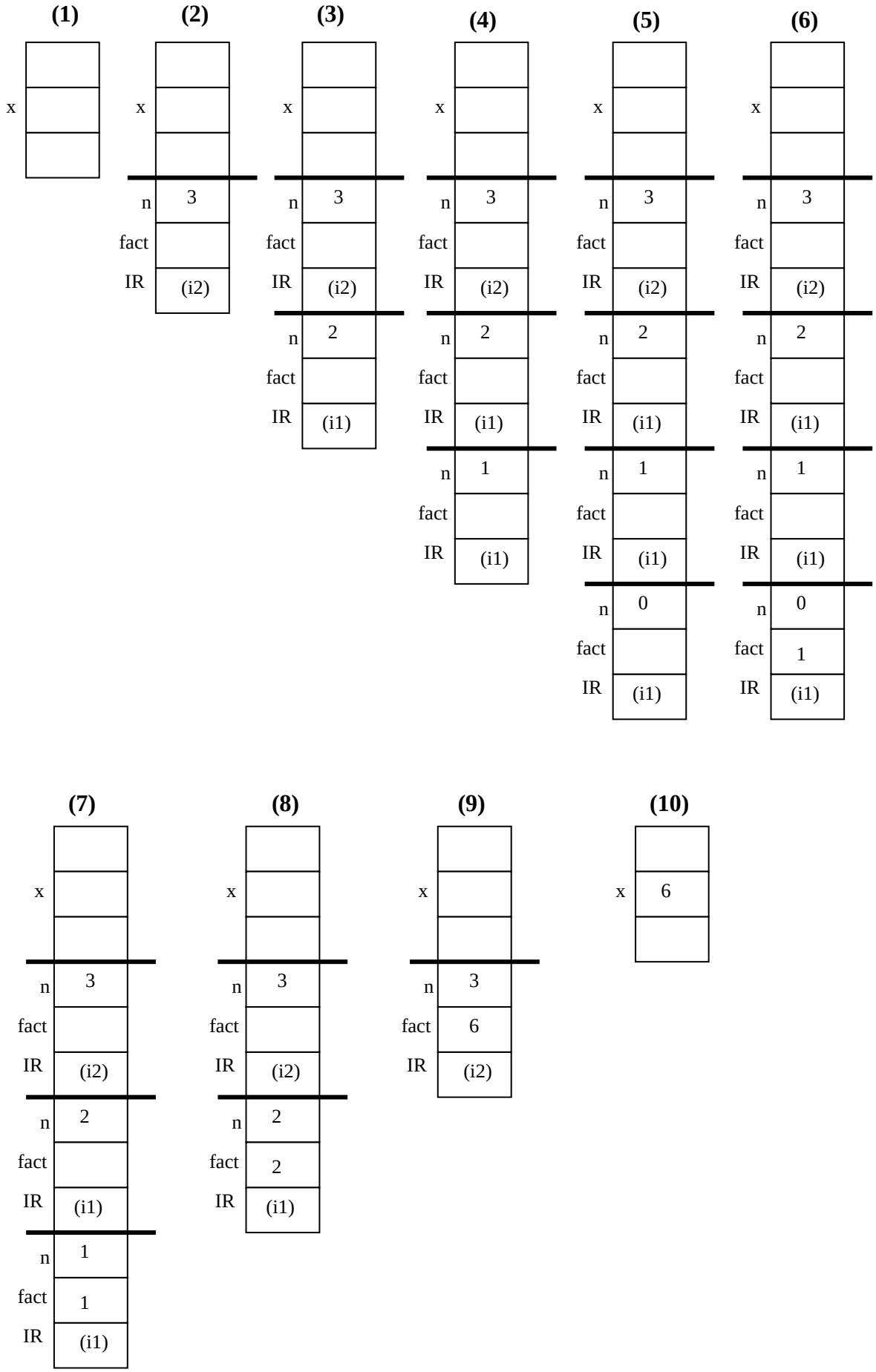
Nel seguito, commenteremo brevemente i vari passi nella valutazione di x=fact(3), facendo riferimento alla Figura 3.1.

(1) descrive la situazione prima della chiamata. Il record di attivazione del main contiene diverse celle (per le variabili globali), di cui una per la variabile x.
(2) All'atto della chiamata a fact(3), viene allocato il RA per la funzione fact. Questa contiene una cella per il parametro n, una per il risultato della funzione (indicata con fact) ed una per l'IR. Prima dell'esecuzione della funzione vengono passati i parametri attuali e legati ai parametri formali. In questo caso, il valore 3 viene copiato (passaggio parametri per valore) nella cella n. L'IR e' l'indirizzo dell'assegnamento x=fact(3) nel main (indicato con (i2)).
(3) Poiche' n>0, l'esecuzione del corpo della funzione comporta l'esecuzione di n*fact(n-1), e quindi una chiamata ricorsiva di fact sul valore 2. Questo comporta l'allocazione di un nuovo RA, ed il passaggio parametri. IR e' (i1), ovvero, l'indirizzo dell'operazione di "*". Si noti che tale operazione viene "lasciata in sospeso", in quanto potra' essere valutata solo quando sara' disponibile il risultato della chiamata ricorsiva.
(4) Poiche' n>0, l'esecuzione del corpo della funzione comporta l'esecuzione di n*fact(n-1), e quindi una chiamata ricorsiva di fact sul valore 1. Questo comporta l'allocazione di un nuovo RA, ed il passaggio parametri (come al passo (3)).
(5) Poiche' n>0, l'esecuzione del corpo della funzione comporta l'esecuzione di n*fact(n-1), e quindi una chiamata ricorsiva di fact sul valore 0. Questo comporta l'allocazione di un nuovo RA, ed il passaggio parametri (come ai passi (3),(4)).
(6) Qui n=0, ed abbiamo quindi raggiunto il passo base della ricorsione (dell'induzione). Viene eseguita l'istruzione return 1, e l'esecuzione di questa chiamata di fact termina.
(7) Il RA per la chiamata fact(0) viene disallocato, e si sitorna nell'ambiente chiamante all'esecuzione dell'istruzione indicata nel RA appena disallocato. Si noti pero' che all'atto della disallocazione del RA di una funzione viene ance (automaticamente) eseguita la restituzione del valore della funzione (in questo caso: fact(0)=1) all'ambiente chiamante. Viene quindi eseguito il prodotto n*fact(0). In questo ambiente, n e' uguale ad 1, e quindi si ha 1*1=1. Tale valore viene inserito nella cella fact che contiene il risultato della funzione (restituzione del valore finale di una funzione), e l'esecuzione di questa chiamata di fact termina.
(8) Il RA per la chiamata fact(1) viene disallocato, e si sitorna nell'ambiente chiamante all'esecuzione dell'istruzione indicata nel RA appena disallocato, restituendo inoltre al chiamante il risultato della funzione chiamata (in questo caso, 1). In questo ambiente, n=2, e quindi n* il risultato della chiamata da' 2. Questo valore viene inserito nella cella fact che contiene il risultato della funzione (restituzione del valore finale di una funzione), e l'esecuzione di questa chiamata di fact termina ritornando tale valore all'ambiente chiamante.
(9) Analogo al precedente, con n=3, fact(2)=2, e quindi il risultato restituito e' 3*2=6.
(10) Il RA di fact(3) viene disallocato, e l'esecuzione della chiamata fact(3) termina, con restituzione del valore di fact(3) al chiamante. Viene eseguito l'assegnamento ad x del risultato restituito dalla funzione.

Nota: E' importante notare come in questo caso (come del resto in tutti i casi di ricorsione non di coda), vengono lasciate delle operazioni "in sospeso", che possono essere completate solo una volta terminata la chiamata ricorsiva. Il calcolo eseguito precedentemente puo' essere infatti schematizzato come mostrato nel seguito. Si noti che solo una volta ottenuto il valore di base della computazione dal passo base della ricorsione (ovvero, fact(0) ritorna il valore 1) verranno chiuse le operazione lasciate in sospeso, nel seguente ordine.

```
fact(3)= 3*fact(2)
          fact(2)= 2*fact(1)
                    fact(1)= 1*fact(0)
                              fact(0)=1
                    fact(1)=1*0 (ove 0=fact(0))
         fact(2)=2*1 (ove 1=fact(1))
fact(3)=3*2 (ove 2=fact(2))
```

Figura 3.1 Fattoriale, ricorsivo non di coda



COMPLESSITA' delle funzioni RICORSIVE

Anche solo considerando questo primo esempio, e' possible notare una grande differenza fra le funzioni iterative e quelle ricorsive. Nella sua definizione iterativa, l'esecuzione della funzione fattoriale esegue O(n) operazioni (prodotto e sottrazione), ed utilizza un record di attivazione. Nella versione ricorsiva le operazioni rimangono in prima approssimazione le stesse (in realta', a livello di assembler, ogni esecuzione di una chiamata di funzione implica diverse operazioni, quali ad esempio l'allocazione di un nuovo record di attivazione, il passaggio parametri, ...), ma non e' piu' sufficiente utilizzare un solo record di attivazione: viene allocato un record di attivazione per ogni richiamo ricorsivo.

Al fine di studiare la complessita', considerando anche funzioni ricorsive, occorre precisare quindi meglio questa nozione.
In particolare, la **complessita' in TEMPO** valuta come nel caso iterativo il tempo necessario ad eseguire la funzione. Nel caso di funzioni ricorsive, spesso tale tempo puo' essere approssimato dal numero di chiamate alla funzione necessarie per risolvere il problema.

Definizione: complessita' in tempo di un programma ricorsivo.
La complessita' in tempo di un programma ricorsivo e' data dal numero di chiamate ricorsive eseguite dal programma stesso.

Ad esempio, nel caso del fattoriale, al fine di valutare il fattoriale di n la funzione fact viene richiamata n+1 volte, e la complessita' in tempo e' quindi O(n).

Nel caso di funzioni ricorsive, tuttavia, ad ogni richiamo ricorsivo viene allocato un nuovo record di attivazione. Quindi, anche lo SPAZIO necessario per eseguire una funzione deve essere considerato. Si parla quindi anche di **complessita' in SPAZIO**, che, come per quella in tempo, deve essere rapportata all'input.

Definizione: complessita' in spazio di un programma ricorsivo.
La complessita' in spazio di un programma ricorsivo e' data dal massimo numero di record di attivazione che possono essere contemporaneamente presenti sulla stack.

Ad esempio, nel caso del fattoriale, al fine di valutare il fattoriale di n la funzione fact alloca in memoria n+1 record di attivazione, e la complessita' in spazio e' quindi O(n).

Nel caso del fattoriale, i record di attivazione vengono prima allocati fino a raggiungere il passo base, e poi disallocati. In casi piu' complessi, tuttavia, il numero di record di attivazione sulla pila puo' variare crescendo e decrescendo piu' volte durante l'esecuzione della funzione (si consideri, ad esempio, la soluzione ricorsiva del problema delle "Torri di Hanoi", nel seguito – Sezione 3.4). Al fine di determinare la quantita' di memoria utilizzata (al massimo) da una funzione ricorsiva, e' quindi importante determinare non tanto quanti record di attivazione siano allocati da una funzione, ma piuttosto **il numero massimo di record di attivazione contemporaneamente allocati da tale funzione** (ovvero: la massima occupazione della pila di esecuzione).

Nota1: In generale, nel caso iterativo (non ricorsivo), ogni funzione alloca un unico record di attivazione (a meno che, ovviamente, queste non contengano al loro interno chiamate ad altre funzioni). Si confronti ad esempio la versione ricorsiva e quella iterativa della funzione fattoriale.

Nota2: Non sempre la complessita' in tempo e spazio delle funzioni ricorsive sono uguali. Ad esempio, vedremo nella Sezione 6 che la complessita' in tempo della funzione Mergesort di ordinamento e' O(n log n), mentre la sua complessita' in spazio e' O(log n).

3.2 RICORSIONE NUMERICA E SU VETTORI

Al fine di familiarizzare con la nozione di ricorsione, vengono qui mostrati alcuni facili esempi di definizione ricorsiva di funzioni numeriche, e di semplici funzioni operandi su vettori monodimensionali.

```
/* funzione che restituisce la somma di x ed y, supponendo di poter utilizzare solo operazioni di incremento e decremento
ad 1. In tale ipotesi, x+y corrisponde a sommare y volte 1 ad x, cioe' x+y=x+1+1+…+1 (y volte))*/
int somma1(int x, int y)
{
  if (y==0) return x;
  else return 1+somma1(x,y-1);
}
/* passo base: somma1(x,0)=x */
/* ipotesi induttiva: somma1(x,y-1) restituisce x+(y-1) */
/* passo induttivo: x+y = (x+(y-1))+1 */
/* terminazione: ad ogni richiamo, y e' decrementata di uno, fino a raggiungere
il valore 0 del passo base */
/* complessita':
       tempo: O(n)
       spazio: O(n) */


/* funzione che restituisce il prodotto di x ed y, supponendo di poter utilizzare solo operazioni di somma e differenza */
int prod(int x, int y)
{
  if (y==0) return 0;
  else return x+prod(x,y-1);
}


/* funzione che restituisce x elevato ad y, supponendo di poter utilizzare solo operazioni di somma, prodotto e differenza */
int exp(int x, int y)
{
  if (y==0) return 1;
  else return x*exp(x,y-1);
}


/* funzione che restituisce la parte intera di x/ y, supponendo di poter utilizzare solo operazioni di somma e differenza */
int div1(int x, int y)
{
  if (x<y) return 0;
  else return 1+div1(x-y,y);
}


/* funzione che restituisce il resto di x/ y, supponendo di poter utilizzare solo operazioni di somma e differenza */
int mod(int x, int y)
{
  if (x<y) return x;
  else return mod(x-y,y);
}
```

```
/* valutare la somma di tutti gli elementi di un array ad n+1 elementi, in posizione da 0 ad n */
int sumA(int A[], int n)
{
  if (n<0) return 0;
  else return A[n]+sumA(A,n-1);
}
/* passo base: la somma degli elementi dell'array da A[n] ad A[0] e' zero,
se n<0 */
/* ipotesi induttiva: sumA(A,i) restituisce la somma A[0]+A[1]+ ...+A[i] */
/* passo induttivo: sumA(A,i) = A[i] + la somma A[0]+A[1]+ ...+A[i-1] */
/* terminazione: ad ogni richiamo, n e' decrementata di uno, fino a raggiungere
il valore minore di 0 del passo base */
/* complessita'
        tempo: O(n)
        spazio: O(n) */


/* stampa ricorsiva di un array di n+1 interi */
void printarr_rt(int A[], int n, int pos)
/* pos inizialmente e' 0 */
{
  if (pos <= n)
  {
    printf("%d\n",A[pos]);
    printarr_rt(A, n, pos+1);
  }
}

/* stampa un array di n+1 interi in ordine inverso */
void printarr_rn(int A[], int n, int pos)
/* n e' il numero di elemeti dell'array A */
/* pos inizialmente e' 0 */
{
  if (pos <= n)
  {
    printarr_rn(A, n, pos+1);
    printf("%d\n",A[pos]);
  }
}


/* sommatoria degli elementi di un array (a n+1 elementi) in posizione multipla di pos */
int sumarrpos_rn(int A[], int n, int pos, int curpos)
/* la prima posizione pos=1 e' A[0] */
/* curpos indica la posiz. corrente, ed e' inizializzata a 1 */
{
  if (curpos>n+1) return 0;
  else if ((curpos % pos) == 0)
                    return A[curpos-1]+sumarrpos_rn(A,n,pos,curpos+1);
     else return sumarrpos_rn(A,n,pos,curpos+1);
}
```

```
/* sommatoria degli elementi di un array (a n elementi) in posizione multipla di pos */
/* versione piu' efficiente: avanza di pos in pos */
/* Complessita': (n+1) / pos */
int sumarrpos2_rn(int A[], int n, int pos, int curpos)
/* curpos inizializzata a pos */
{
  if (curpos>n+1) return 0;
  else return A[curpos-1]+sumarrpos2_rn(A,n,pos,curpos+pos);
}

/* restituisce il numero di elementi dell'array che sono multipli di x */
int countmul_rn(int A[], int n, int x)
{
  if (n<0) return 0;
  else if ((A[n] % x) == 0) return 1+countmul_rn(A,n-1,x);
      else return countmul_rn(A,n-1,x);
}

/* restituisce 1 se A e' una palindrome, 0 altrim. */
/* NOTA: una palindrome e' una sequenza –di numeri, o di caratteri- che risulta uguale se letta da sinistra a destra o letta da
destra a sinistra. Ad esempio, 1,5,6,7,6,5,1 e' una palindrome, cosi' come 2,4,6,6,4,2 */
int palind_rn(int A[], int n, int i)
/* i inizializzata a 0 */
{
  if (i > (n-i))
            return 1;
  else if (A[i] != A[n-i]) return 0;
  else return palind_rn(A,n,i+1);
}

/* funzione ausiliaria */
int min(int x, int y)
{
  if (x<y) return x;
  else return y;
}

/* restituisce il minimo di un array di n elementi */
int minarr_rn(int A[], int n)
{
  if (n==0) return A[0];
  else return min(A[n],minarr_rn(A,n-1));
}
```

```
/* sottrae x a tutti gli elementi dell'array multipli di x */
void dividix(int A[], int n, int x)
{
  if (n>=0)
            {
             if ((A[n] % x) == 0) A[n]=A[n]-x;
             dividix(A,n-1,x);
            }
}

/* conta le occorrenze di x in A */
int countx_rn(int A[], int n, int x)
{
  if (n<0) return 0;
  else if (A[n]==x) return 1+countx_rn(A,n-1,x);
            else return countx_rn(A,n-1,x);
}
```

3.3 TIPI DI RICORSIONE (CENNI)

Finora e' stato considerate un unico e molto semplice tipo di ricorsione: la ricorsione lineare non di coda. Piu' in generale, e' possibile distinguere tra differenti tipi di programmi ricorsivi, a seconda del tipo di ricorsione utilizzata.
Anzitutto, nel presente corso ci limiteremo a trattare la **ricorsione "diretta"**, ovvero funzioni ricorsive che richiamano direttamente se stesse (come ad esempio le funzioni *fact*, *somma1* e *sumA* viste in precedenza). E' comunque importante ricordare che, in taluni casi, si potrebbe avere anche una ricorsione "indiretta", detta normalmente **mutua ricorsione**. Si ha una mutua ricorsione quando ad esempio una funzione A richiama al suo interno una funzione B e, a sua volta, B richiama al suo interno A, come esemplificato nel seguito.

Esempio di mutua ricorsione:
```
A(...) { ..... B(....); ..... }
B(...) { ..... A(....); ..... }
```

Nel seguito della trattazione considereremo solo la ricorsione diretta.

Definizione: ricorsione lineare.
Una funzione e' detta **ricorsiva lineare** se comporta al piu' un unico richiamo ricorsivo.

Nota: Si noti che questo non vuol dire che la funzione contiene al suo interno un'unica chiamata ricorsiva ma piuttosto che per ogni possibile esecuzione della funzione si esegue al piu' una chiamata ricorsiva. Si consideri ad esempio la seguente traccia di funzione, in cui sono evidenziate tutte le chiamate ricorsive:

```
void A(int x, ...)
{
        if (x==0) {…}
        else if (x==1) { .....A(.....); .....}
        else { ... A(....); .. }
```

Nel testo della funzione compaiono due chiamate ricorsive, ma non piu' di una di esse puo' essere eseguita per ogni chiamata della funzione A, che e' quindi ricorsiva lineare.

Le funzioni finora presentate (sezione 3.1 e 3.2) sono tutte ricorsive lineari, mentre ad esempio la funzione delle torri di Hanoi nel seguito non e' ricorsiva lineare.

Definizione: ricorsione di coda (tail recursion).
Una chiamata ricorsiva viene detta ricorsiva di coda (tail recursive) se e' una chiamata terminale, ovvero se essa e' l'ultima istruzione o l'ultima operazione eseguita dal programma chiamante (ovvero, se al ritorno nel programma chiamante rimane solo da eseguire l'"end" del programma, al piu' preceduto dalla restituzione del risultato tramite una return).

Definizione: funzioni ricorsive di coda.
Una funzione ricorsiva e' ricorsiva di coda se e' ricorsiva lineare e ogni possible chiamata ricorsiva in essa contenuta e' ricorsiva di coda.

Definizione: funzioni ricorsive non di coda.
Una funzione o funzione ricorsiva e' ricorsiva non di coda se e' ricorsiva lineare e almeno una possible chiamata in essa contenuta non e' di coda (ovvero, deve essere seguita da altre operazioni prima della chiusura del recod di attivazione).

Si noti che non necessariamente una funzione ricorsiva di coda e' scritta in modo tale da terminare con una chiamata ricorsiva. Ad esempio, la seguente traccia di funzione, in cui sono evidenziate tutte le chiamate ricorsive, e' ricorsiva di coda (in quanto, in entrambi i casi in cui la chiamata ricorsiva puo' essere eseguita, tale esecuzione e' l'ultima operazione della funzione).

```
void A(int x, ...)
{
        if (x==0) {……}
        else if (x==1) { .....A(.....); }
```

```
else if (x==2) { ... A(....);}
else { ..... }
```

Al contrario, e' importante notare che il fatto che la chiamata ricorsiva sia scritta come ultima istruzione di una funzione non vuol assolutamente dire che tale funzione si ricorsiva di coda.
Ad esempio, la funzione fattoriale , riportata qui per chiarezza, NON e' ricorsiva di coda, in quanto dopo l'esecuzione della chiamata ricorsiva "fact(n-1)" la funzione chiamante deve ancora eseguire una operazione, ovvero il prodotto di n per il risultato della chiamata ricorsiva, prima di assegnare il valore a fact, restituendo cosi' il risultato al chiamante.

```
int fact(int n)
{
  if (n==0)return 1;
  else return n*fact(n-1);
}
```

Nota: Tutte le funzioni ricorsive di coda sono per definizione funzioni ricorsive lineari. Non vale il viceversa (ad esempio, la funzione fact e' ricorsiva lineare ma non ricorsiva di coda).

Nota: ove e' sato inserito, il suffisso _XX dei nomi di funzioni indica il tipo di ricorsione adottata:
_rt indica la ricorsione tail (di coda)
_rn indica la ricorsione non tail (non di coda)

NOTA: L'esecuzione di una funzione iterativa richiede un unico record di attivazione. Al contrario, l'esecuzione di una funzione ricorsiva richiede una stack di record di attivazione (ad esempio, nel caso di fact(n), sono richiesti n+1 record di attivazione). Ne consegue una maggiore efficienza in spazio di funzioni iterative.
Nel caso delle funzioni ricorsive di coda, tuttavia, l'uso della stack non e' strettamente indispensabile, in quanto il risultato viene calcolato incrementalmente ad ogni passo, per cui al passo base si ottiene il risultato, che puo' essere restituito al chiamante direttamente. Per tale motivo, e' possibile trasformare le funzioni ricorsive di coda in funzioni iterative.
In tale ottica, e'possibile definire le seguenti trasformazioni (che non verranno tuttavia sviluppate all'interno del presente corso):
1) da funzione ricorsiva lineare a ricorsiva di coda
2) da funzione ricorsiva di coda ad iterativa

Anche se non viene mostrato il procedimento di trasformazione da ricorsione non di coda a ricorsione di coda, vengono nel seguito mostrati alcuni esempi di ricorsioni non di coda. In particolare, si consideri la funzione fattoriale nel seguito, che utilizza un parametro per riferimento in modo da "accumulare" passo passo i risultati del prodotto, che viene eseguito subito, anziche' lasciato in sospeso come nella definizione non di coda.

```
int fact'(int n, int *acc)
{
      if (n==0) return (*acc);
      else
            {
            (*acc)=(*acc)*n;
            return fact'(n-1,acc)
            }
}

int fact_t(int n)
{int ris =1;
 return fact'(n,&ris);
}
```

Nel seguito, commenteremo brevemente i vari passi nella valutazione di x:=fact_t(3), facendo riferimento alla Figura 3.2.

(1) descrive la situazione prima della chiamata di fact'. Il primo RA e' quello relativo al main, e contiene una cella per la variabile x. Il secondo RA in stack e' quello relativo alla funzione fact_t, che ha parametro n=3 e variabile locale ris inizializzata ad 1. Come prima, (i2) indica l'indirizzo dell'assegnamento x=fact_t(3) nel main. In questa situazione viene richiamata la funzione fact' con parametri attuali n e ris.

(2) All'atto della chiamata a fact'(n,ris), viene allocato il RA per la funzione fact'. Questa contiene una cella per il parametro n, una per acc, una per il risultato della funzione (indicata con fact') ed una per l'IR. Prima dell'esecuzione della funzione vengono passati i parametri attuali e legati ai parametri formali. In questo caso, il valore 3 viene copiato (passaggio parametri per valore) nella cella n, mentre acc e' un parametro per riferimento corrispondente alla variabile ris (questo e' indicato con @ris in figura 2-2-2. L'IR e' l'indirizzo dell'istruzione return fact'(n,ris) (indicato con (i0)).

(3) Poiche' n>0, l'esecuzione del corpo della funzione fact' comporta l'esecuzione di (*acc)=(*acc)*n, che valuta 1*3=3 ed assegna il risultato ad acc. Poiche' acc e' un riferimento a ris, il valore 3 viene messo nella cella ris. Quindi viene eseguita una chiamata ricorsiva di fact' su n-1=2 ed il nuovo valore di acc. Questo comporta l'allocazione di un nuovo RA, ed il passaggio parametri. IR e' (i1), ovvero, l'indirizzo dell'operazione di return fact'(n-1,acc), che termina la chiamata ricorsiva alla funzione fact' ritornando il valore ottenuto nella chiamata stessa.

(4) Poiche' n>0, l'esecuzione del corpo della funzione comporta l'esecuzione di di (*acc)=(*acc)*n, che pone 3*2=6 nella cella riferita da acc (ovvero in ris). Quindi viene eseguita una chiamata ricorsiva di fact' su n-1=1 ed acc. Questo comporta l'allocazione di un nuovo RA, ed il passaggio parametri. (come al passo (3)).

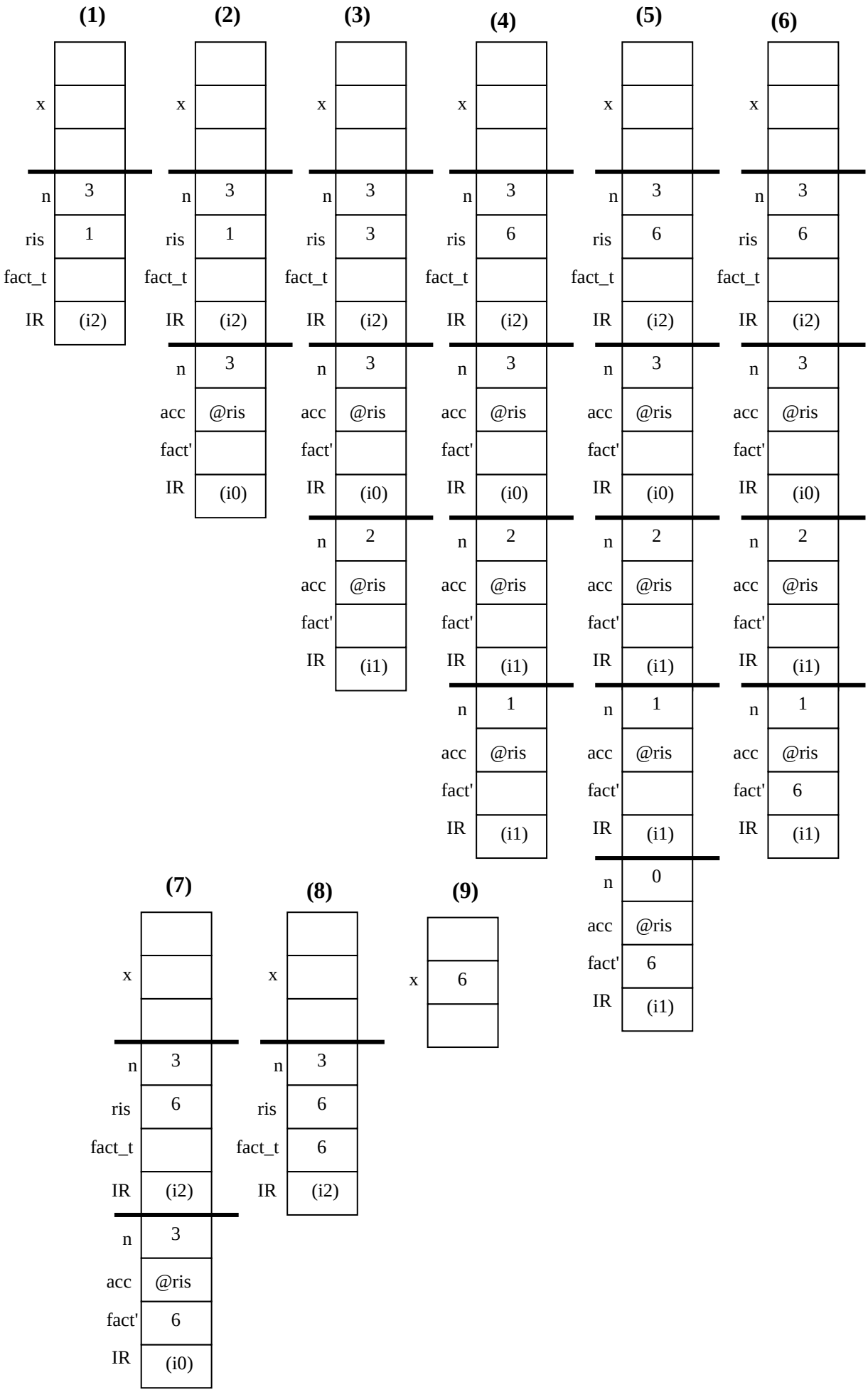
(5) Poiche' n>0, l'esecuzione del corpo della funzione comporta l'esecuzione di (*acc)=(*acc)*n, che pone 6*1=6 nella cella riferita da acc (ovvero in ris). Quindi viene eseguita una chiamata ricorsiva di fact' su n-1=0 ed acc. Questo comporta l'allocazione di un nuovo RA, ed il passaggio parametri (come ai passi (3),(4)). Qui n=0, ed abbiamo quindi raggiunto il passo base della ricorsione (dell'induzione). Viene eseguita l'istruzione “return (*acc)”, e l'esecuzione di questa chiamata di fact' termina.

(6) I passi successivi (alcuni passi sono omessi in figura) si limitano a disallocare uno per volta i RA, restituendo sempre al chiamante il valore del risultato computato dalla chiamata ricorsiva. Ad esempio, (6) mostra la situazione dopo che e' disallocato l'ultimo RA, e (7) dopo la disallocazione di altri due RA.

Nota: E' importante notare come in questo caso (come del resto in tutti i casi di ricorsione non di coda), non vengono lasciate delle operazioni "in sospeso": i prodotti sono eseguiti appena possibile, cosicche' quando si arriva al passo base della funzione ricorsiva (ovvero dell'induzione; caso n=0) si ha direttamente il risultato finale, senza dover attendere la chiusura delle chiamate ricorsive (che hanno il solo scopo di trasmettere al chiamante il risultato cosi' ottenuto).

Nota2: Per omogeneita' con la funzione fact ricorsiva non di coda vista in precedenza, si e' scelto di realizzare fact' come una funzione. In realta', in questo caso risulterebbe piu' semplice la definizione di una funzione come fact''.

Figura 3.2 Fattoriale ricorsivo di coda



Si noti che, in precedenza, e' stata introdotta la funzione fact_t al fine di "nascondere" il parametro per riferimento aggiuntivo necessario per ottenere una valutazione"di coda". Se invece si puo'/vuole lasciare esplicito tale parametro, la funzione fattoriale ricorsiva di coda puo' essere direttamente definita come segue:

```
void fact"(int n, int *acc)
/* *acc inizializzato ad 1 */
{
  if (n>0)
    {
          (*acc)=n*(*acc);
          fact"(n-1,acc);
    }
}
```

Nel seguito viene mostrata la versione "di coda" di alcune delle funzioni ricorsive precedentemente mostrate nella sezione 3.2.

/* funzione che restituisce la somma di x ed y, supponendo di poter utilizzare solo operazioni di incremento e decremento ad 1. In tale ipotesi, x+y corrisponde a sommare y volte 1 ad x, cioe' x+y=x+1+1+…+1 (y volte +1))
Versione DI CODA, con il parametro per riferimento inizializzato a zero */

```
void somma1_rt(int x, int y, int *ris)
/* ris inizializzato a x */
{
  if (y>0)
          {
                  *ris=*ris + 1:
                   somma1(x,y-1,ris);
          }
}
```

/* somatoria degli elementi di un array (a n elementi) in un parametro ris */

```
void sumarr_rt(int A[], int n, int *ris)
/* ris inizializzato a 0 */
{
  if (n>=0)
  {
   *ris+=A[n];
   /* NB occorre incrementare *ris */
   sumarr_rt(A,n-1,ris);
  }
}
```

3.4 UN ESEMPIO COMPLESSO DI RICORSIONE: le torri di Hanoi

Il problema delle Torri di Hanoi e' un problema "classico" dell'informatica, che ben si presta a esemplificare la "potenza" del metodo ricorsivo.
In sintesi, il problema e' il seguente. Si supponga di avere 3 pioli (chiamati ad esempio A, B, e C), e n (n intero e maggiore di zero) dischi, di grandezze tutte diverse. Si supponga quindi che tutti i dischi siano stati infilati ad un piolo (il piolo A), in ordine di grandezza crescente (ovvero, il disco piu' grande e' in fondo, …, e il disco piu' piccolo e' in cima alla pila). L'obiettivo e' quello di spostare tutti i dischi dal piolo A al piolo C, ri-impilandoli in ordine ccrescente, e seguendo i seguenti vincoli:
(1) Si puo' muovere un solo disco alla volta, e
(2) Nessun disco puo' mai essere posizionato sopra ad un disco piu' piccolo

Viene richiesta di stampare in output la sequenza di tutte (e sole) le mosse necessarie a raggiungere il risultato. Per ogni mossa, e' sufficiente indicare chi e' il piolo di origine (il piolo da cui viene preso il disco piu' in alto fra quelli impilati) e chi quello di arrivo (ovvero, il piolo su cui viene impilato tale disco).

Si noti che, con n>3, non e' banale per una persona trovare una soluzione, ed il problema diventa tanto piu' complesso quuanto piu' cresce n. Poi, nel caso specifico, n non e' nota, ma e' uno dei parametri in input. Definire la soluzione iterativa significa essere in grado di determinare la sequenza delle mosse necessarie, dando un algoritmo esplicito per il calcolo di tali mosse.
La ricorsione, invece, ci fornisce uno strumento molto piu' semplice per la gestione di questo problema, dal momento che ci permette di avvalerci delle "ipotesi induttive": qualora siamo in grado di suddividere il problema in uno o piu' sottoproblemi uguali a quello di partenza, ma "piu' piccoli", non dobbiamo precisare come risolvere tali sottoproblemi, ma possiamo avvalerci dell'ipotesi induttiva.
La soluzione ricorsiva e' semplice, proprio per questo motivo. E' la funzione move nel seguito, che ha quattro parametri: il numero di dischi da spostare, ed i tre pioli (nell'ordine: piolo di partenza, piolo obiettivo, e piolo usato come supporto negli spostamenti). La funzione e' inizialmente richiamata come *move(n,A,B,C*).

Il passo base e' banale: e' il caso di un solo disco. Se ho un solo disco, mi e' sufficiente muoverlo direttamente dal piolo A al piolo C, ed il problema e' risolto.

Se invece i dischi sono n (n>1), per determinatre la soluzione ricorsiva, devo essere in grado di suddividere il problema in problemi simili, ma piu' piccoli (ovvero, con meno dischi da muovere, in questo esempio). Il ragionamento e' ssemplce: se io devo muovere n dischi da piolo (diciamo start) ad un altro (diciamo end) usandone un terzo (temp) come supporto, dati i vincoli (1) e (2), e' chiaro che il primo disco che devo spostare e' il piu' grande disco su start, che deve essere messo alla bare di end. Ora, per fare questo, e' necessario prima togliere tutti i dischi in start ch sono sopra al piu' grande (ovvero, n-1 dischi). Questi dischi devono ovviamente essere messi su temp, perche' voglio poter mettere il disco piu' grande su end (e non posso metterlo sopra a dischi piu' piccoli di lui). Quindii, necessariamente, devo operare in tre passi:
(1) Devo spostare n-1 dischi (tutti tranne il piu' grande) da start a temp (usando end come supporto)
(2) Devo spostare un disco da start ad end
(3) Devo spostare n-1 dischi da temp ad end (usando start come supporto)

Ora, il passo (2) e' chiaramentte una operazione di base (stampo la singola mossa che devo fare). Di per se, i passi (1) e (3) possono essere molto ccomplessi (se n e' grande), ma attenzione: il problema da affrontare al passo (1) e quello al passo (3) sono esattamente come il problema di partenza, con una differenza irrilevante (sto scambiando chi sono i pioli di partenza, di arrivo, e di supporto), ed una fondamentale: operano su un disco in meno! E' quindi corretto applicare "l'ipotesi indutttiva", e risolverli mediante una chiamata ricorsiva alla funzione "move". Si ottiene quindi facilmente l'algoritmo ricorsivo mostrato nel seguito.

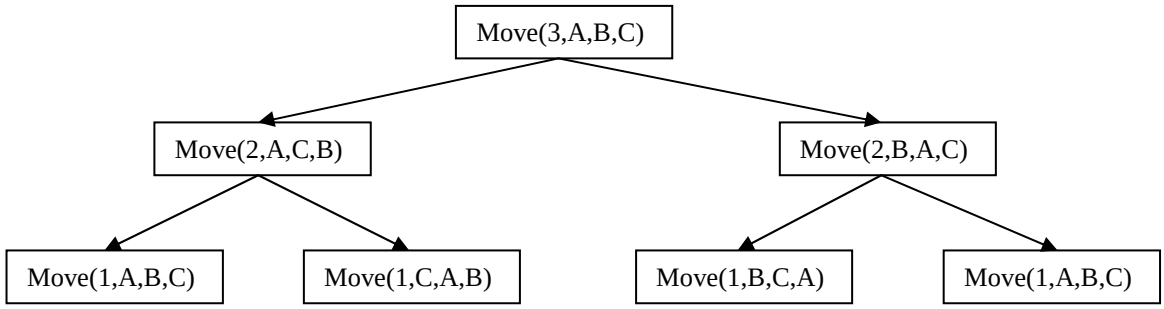
```
void move(int n, char A, char B, char C)
{
if (n==1) prinf("%s%c%s%c\n", "move from tower", A, "to tower", C);
else
        {
        move(n-1,A,C,B);
        prinf("%s%c%s%c\n", "move from tower", A, "to tower", C);
        move(n-1,B,A,C);
        }
}
```

Richiamo della funzione (nel main): **move(n,A,B,C)**

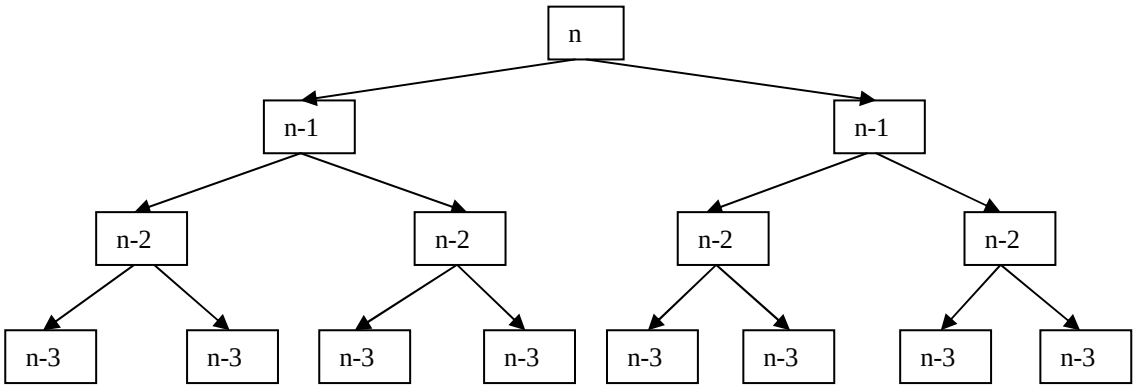
Analisi e complessita’

L’algoritmo proposto adotta una tecnica di programmazione molto importante e diffusa: la suddivisione in di problem in sottoproblemi (paradigma “*divide et impera*”). Un problema di dimensione *n* (numero dei dischi da spostare) viene ridotto ricorsivamente ad una operazione atomica (lo spostamento di un disco) e due problemi *dello stesso tipo*, ma piu’ piccoli (spostare *n-1* dischi).

Ad esempio, viene mostrato nel seguito l’albero delle chiamate ricorsive della funzione ***move***, applicata a tre dischi. Si noti che le chiamate formano un albero binario (ovvero, una struttura in cui, a partire da una radice, ogni elemento ha al piu’ due figli). Ogni nodo dell’albero rappresenta una chiamata della funzione *move*, e corrisponde quindi ad un record di attivazione.



Considerando piu’ in generale l’applicazione di *move* ad *n* dischi, ed evidenziando solo il numero di dischi cui si applica ogni chiamata, si ottiene il seguente albero di chiamate (o, equivalentemente, di record di attivazione)



.....

Analisi: Dalla seconda figura, si evince che il numero delle chiamate cresce esponenzialmente ad ogni livello:

- Livello 1: 1 chiamata (su n dischi)
- Livello 2: 2 chiamate (su n-1 dischi)
- Livello 3: 4 chiamate (su n-2 dischi)
- Livello 4: 8 chiamate (su n-3 dischi)
- ....
- Livello k: 2^{k-1} chiamate (su n-k-1 dischi)

Dal momento che, ad ogni livello, il richiamo avviene su un disco in meno, e che il passo base lavora su un disco, l'albero delle chiamate ha ***n* livelli**.

Complessita' in tempo. Ad ogni chiamata della funzione *move*, viene esguito esattamente lo spostamento di un disco. La complessita' in tempo puo' quindi essere misurata, equivalentemente, dal numero di chiamate o dal numero degli spostamenti. Data l'analisi precedente, e' evidente che il numero delle chiamate e'

$$2^0+2^1+2^2+2^3+ ...+2^{n-1} = 2^n-1$$

La complessita' in tempo e' quindi esponenziale: $\mathbf{O(2^n)}$

Complessita' in spazio. Considerando una costante lo spazio occupato dal record di attivazione per eseguire una chiamata di move, la complessita' in spazio (caso peggiore) e' data dal massimo numero di record di attivazione contemporaneamente presenti sulla stack. Si noti che, anche se ad ogni chiamata di *move* viene allocato un record di attivazione, la stack dei record cresce e decresce durante l'esecuzione.
Per esempio, nel seguito viene mostrata l'evoluzione della stack corrispondente al richiamo della funzione *move* su tre dischi (si veda l'albero delle chiamate nelle figure precedenti). Ogni riga corrisponde ad una configurazione della stack, e si suppone che la stack cresca da sinistra a destra

Move(3,A,C,B)
Move(3,A,C,B) || Move(2,A,B,C)
Move(3,A,C,B) || Move(2,A,B,C) || Move(1,A,C,B)
Move(3,A,C,B) || Move(2,A,B,C)
Move(3,A,C,B) || Move(2,A,B,C) || Move(1,C,B,A)
Move(3,A,C,B) || Move(2,A,B,C)
Move(3,A,C,B)
Move(3,A,C,B) || Move(2,B,C,A)
Move(3,A,C,B) || Move(2,B,C,A) || Move(1,B,A,C)
Move(3,A,C,B) || Move(2,B,C,A)
Move(3,A,C,B) || Move(2,B,C,A) || Move(1,A,C,B)
Move(3,A,C,B) || Move(2,B,C,A)
Move(3,A,C,B)

In generale, si avranno quindi sulla stack al piu' un numero di record di attivazione pari al numero dei livelli dell'albero delle chiamate. La complessita' in spazio e' quindi lineare: $\mathbf{O(n)}$