NOTE aggiuntive al corso di Programmazione II
Docente: Paolo Terenziani
a.a. 2000-2001

Le note nel seguito costituiscono una integrazione del libro di testo "Data Structures and Program Design in C", Robert L. Cruse, Clovis L. Tondo, Bruce P. Leung, II edizione, Prentice Hall International Editions.

Avvertenza. Sebbene gli algoritmi qui riportati siano stati prima verificati su computer, non si esclude la presenza di errori, dovuti per lo piu' a distrazioni all'atto della trascrizione degli stessi. Gli studenti che rilevassero eventuali errori sono pregati di segnalarli, al fine di migliorare la qualita' delle note stesse.

Parte 0: Strutture in C

0.1 Strutture

Le Strutture sono tipi di dati composti. Nel caso degli array (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};

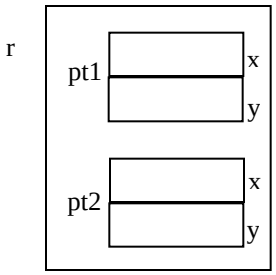
E' necessario inserire tanti valori quanti sono i campi e rispettare i tipi di dati precedentemente dichiarati (cioè se sono stati dichiarati campi di caratteri, non è possibile poi inserire interi).

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;

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);

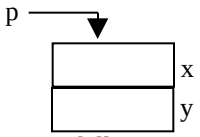
In p ora c'è la struttura con l'ascissa (campo x) che assume il valore x1, e l'ordinata (campo y) che assume il valore x2.

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.

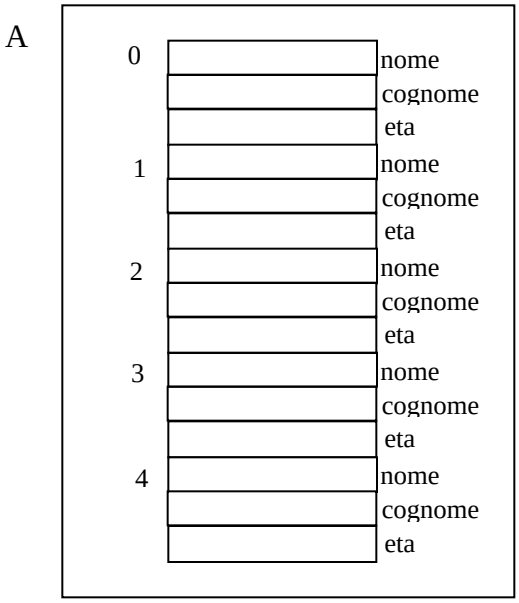
```
void shiftpoint (struct point *p) {
  int num;
  printf("\n ascissa spostata di   ");
  scanf("%d",&num);
  (*p).x+=num; /* è l'equivalente di (*p).x=(*p).x + num */
  printf("\n ordinata spostata di   ");
  scanf("%d",&num);
  (*p).y+=num; /* è l'equivalente di (*p).y=(*p).y + num */
  }
```

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'.



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)++;
      }
    }
}
```

Parte 1: Iterazione su liste (Esercizi)

Le seguenti funzioni prevedono l'inclusione dei seguenti files di libreria:
#include <stdio.h>#include <stdlib.h>
La struttura dati utilizzata nel seguito per rappresentare gli elementi in una lista e' la seguente:

```
typedef struct nod      {       int data;
        struct nod *next;       } node;
```

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.

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

Nel seguito sono proposti alcuni esempi di funzioni.

/* costruzione di una lista */

/* Dato un intero n>0, costruisce la lista di nodi da 1 ad n */
node* buildlis_n (int n){ node *p, *lis; lis=NULL; while (n>0)
/* Invariante: lis e' la lista dei nodi contenenti i, 0<n<i<n,
ove con n si denota il valore di input del parametro "n", e con n il suo valore ad un dato passo del ciclo */
{ p=newnode(); p->data=n; p->next=lis; lis=p; n--; } return(lis); }
Complessita': O(n)

```
/* Costruisce una lista di nodi con .data interi positivi leggendo gli interi da tastiera. 0 indica la terminazione della lista */
node *buildlis()
{
    int x;
    node *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->data = x;
          last->next = NULL;
          printf("nuovo numero da inserire in lista:\n");
          scanf("%d", &x);
          while (x>0)
/* Invariante: lis punta alla lista con tutti gli interi finora letti, tranne l'ultimo, e last punta all'ultimo nodo di tale lista */
          {
               p=newnode();
               p->data = 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. Si noti che per passare come riferimento un puntatore (node *) in C occorre passare un puntatore al puntatore (ovvero (node **lis)
Complessita' ed invariante sono analoghi */

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

Nota1: la maggior parte delle funzioni nel seguito presuppone di ricevere in input una (o piu') lista costruta prima del richiamo della funzione stessa.

/* VISITA DEGLI ELEMENTI DI UNA LISTA */

/* Stampa degli elementi di una lista */
void printlis(node *lis){printf("lista risultato\n");while (lis != NULL)
/* Invariante: stampati tutti i .data dall'inizio della lista fino a lis esclusa */ **{ printf(">>>> %d\n", lis->data); lis= lis->next; }}**
Complessita': O(n) ove n e' il numero di nodi della lista

/* VISITA CON CONDIZIONE */

/* stampa i numeri della lista piu' grandi di x */
void printgreater(node *lis, int x){printf("numeri della lista piu'grandi di %d:\n",x);while (lis != NULL)
/* Invariante: stampati tutti i .data maggiori di x dall'inizio della lista fino a lis esclusa */ **{ if (lis->data > x) {printf(">>>> %d\n", lis->data);} lis= lis->next; }}**Complessita': O(n) ove n e' il numero di nodi della lista

/* VISITA CON CONTATORE E CONDIZIONE */

/* stampa i numeri della lista in posizione multipla di x */
```
void printpos(node *lis, int x)
{
int pos=1;
printf("numeri della lista in posizione multipla di %d:\n",x);
while (lis != NULL)
```
/* Invariante: stampati tutti i .data maggiori di x dall'inizio della lista fino a lis esclusa in posizione multipla di x; pos e' la posizione del nodo lis nella lista lis */
```
    {
        if ((pos % x)==0)
        {printf(">>>> %d\n", lis->data);}
        pos++;
        lis= lis->next;
    }
}
```
Complessita': O(n) ove n e' il numero di nodi della lista

/* stampa il numero della lista in posizione x, se c'e' */
```
void printnth(node *lis, int x){
int pos=1;
```
/* NB alternativamente, si potrebbe direttamente decrementare la x */
```
while ((lis != NULL) && (pos <= x))
    {
```
/* Invariante: pos indica la posizione del nodo puntato da lis nella lista.
pos <=x */
```
        if (pos == x)
        {printf("numero della lista in posizione %d:\n",x);
         printf(">>>> %d\n", lis->data);
         pos++;}
    else
        {pos++; lis= lis->next;}
    }
if (lis == NULL)
printf("nella lista ci sono meno di %d numeri\n",x);}
```
Complessita': O(minimo(n,x)), ove n e' il numero di elementi nella lista

NOTA: Per brevita', nel seguito saranno omessi sia gli invarianti di ciclo che la complessita' delle funzioni proposte.

/* RICERCA DI UN ELEMENTO IN UNA LISTA */

```
/* restituisce un puntatore all'elemento n-esimo di una lista */node *nth(int n, node *p)
/* n => 0 */
{
  while ((n>1)&&(p != NULL))
  {
            p=p->next;
            n--;
  }
  return(p);
}

/* restituisce un puntatore alla prima occorrenza di x in una lista p */
node *find(int x, node *p)
{
    int trovato=0;
    while ((p != NULL) && (! trovato))
       {
            if (p->data==x) trovato=1;
            else p=p->next;
            }
            return(p);
}
```

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

```
node *findnth(int x, int occ, node *lis){int count=0;int trovato=0;while ((lis != NULL) && (!trovato))    {
  lis=find(x,lis);
            if (lis != NULL)
                        if (count+1 == occ)
                                    trovato=1;
                        else
                                    {count=count+1; lis=lis->next;}
            else        lis=NULL;
  }
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 */

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

/* FUNZIONI AGGREGATE SU LISTE */

```
/* sommatoria non condizionata */int sumlis(node *lis)
{
  int acc=0;
  while (lis != NULL)
  {
            acc+=lis->data;
            lis=lis->next;
  }
  return(acc);
}
```

```
/* sommatoria condizionata: somma solo i numeri piu' piccoli di x ed in posizione multipla di n */int sumliscond(node *lis,
int x, int n){
  int acc=0;
  int pos=1;
  while (lis != NULL)
  {
            if (((pos % n)==0) && (lis->data < x))
            acc+=lis->data;
            lis=lis->next;
            pos++;
  }
  return(acc);
}
```

```
/* conta le occorrenze di x in una lista */int countx(int x, node *lis){int acc=0;
while (lis != NULL)
  {
            if (lis->data == x) acc++;
            lis=lis->next;
  }
return acc;
}
```

/* MODIFICA DEL CONTENUTO DEI NODI DI UNA LISTA */

```
/* aumenta di x il contenuto dei nodi in posizione multipla di n, solo se sono piu' grandi di y */
void addcond(node *lis, int x, int y, int n)
{
int pos=1;
/* NB si puo' direttamente decrementare la n */
while (lis != NULL)
    {
        if (((pos % n) == 0) && (lis->data > y)) lis->data+=x;
        pos++;
        lis= lis->next;
    }
}
```

/* MODIFICA DI UNA LISTA: OPERAZIONI SUI LINK */

```
/* inserzione di un nuovo nodo contenente x in testa alla lista lis
prima versione: restituisce un nuovo puntatore alla lista modificata
mentre lis rimane immutato (infatti, e' passata per valore) */
node *headinsert(node *lis, int x)
{
    node *p;
    p=newnode();
    p->data=x;
    p->next=lis;
    return(p);
}
```

```
/* inserzione di un nuovo nodo contenente x in testa alla lista lis
 seconda versione: modificata lis (passata per riferimento) */
void headinsertD(node **lis, int x)
{
    node *p;
    p=newnode();
    p->data=x;
    p->next=*lis;
    *lis=p;
}
```

/* ulteriore possibilita': data una lista l1,
richiamare l1=headinsert(l1,x), cosi' l1 e' comunque modificata */

```
/* inserzione di un nuovo nodo contenente x in coda alla lista x
prima versione: nuovo puntatore alla lista modificata mentre lis rimane immutato (anche se lis==NULL) */
node *tailinsert(node *lis, int x)
{
  node *p, *q;
  q=lis;
  p=newnode();
  p->data=x;
  p->next=NULL;
  if (q == NULL)
  lis=p;
  else
  {
            while (q->next != NULL) q=q->next;
            q->next=p;
  }
  return(lis);
}

/* inserzione di un nuovo nodo contenente x in coda alla lista x; seconda versione: modificata lis */
void tailinsertD(node **lis, int x)
{
  node *p, *q;
  q=*lis;
  p=newnode();
  p->data=x;
  p->next=NULL;
  if (q == NULL)
  *lis=p;
  else
  {
            while (q->next != NULL) q=q->next;
            q->next=p;
  }
}


/* ulteriore possibilita': data una lista l1,  richiamare l1=tailinsert(l1,x), cosi' l1 e' comunque modificata */
```

/* inserzione di newn prima della prima occorrenza di x in lis (se c'e')*/

```
void insertbeforeD(int x, int newn, node **lis)
{
int trovato=0;
node *p,*head;
head=*lis;
if (head == NULL)
  {printf("lista vuota\n");}
else
  {
  if (head->data == x)
    {
            p=newnode();
            p->data=newn;
            p->next=head;
            *lis=p;
     }
  else
    {
            while ((head->next != NULL) && (! trovato))
                    if (head->next->data == x)
                            trovato=1;
                    else
                            head=head->next;
            if (trovato)
                    {
                            p=newnode();
                            p->data=newn;
                            p->next=head->next;
                            head->next=p;
                    }
     }
  }
}
```

```
/* cancellare la prima occorrenza di x in una lista */
void delfirstxD(int x, node **lis)
{
node *p,*q;
if (*lis != NULL)
   if ((*lis)->data==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 delallxD(int x, node **lis)
{
int uguali=1;
node *p,*head;
/* nel primo ciclo, estraggo solo le x che si trovano in testa alla lista */
while ((*lis != NULL)&&(uguali==1))
  {
   if ((*lis)->data==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->data==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 delallxD2(int x, node **lis)
{
node *p,*head;
if (lis != NULL)
  {
    head=*lis;
    while (head->next != NULL)
            {
            if (head->next->data==x)
                    {
                    p=head->next;
                    head->next=head->next->next;
                    free(p);
                    }
            else head=head->next;
            }
    if ((*lis)->data==x) *lis=(*lis)->next;
  }
}
```

/* crea un nuovo nodo contenente x e lo inserisce dopo il nodo l */

```
void insertafternode(int x, node *l)
{
    node *p;
    p=newnode();
    p->data=x;
    p->next=l->next;
    l->next=p;
}
```

/* inserisce un nuovo nodo che contiene new dopo la prima occorrenza di old in lis */

```
void insertafter(int old, int new, node *lis)
{
    node *p;
    p=find(old,lis);
    if (p != NULL) insertafternode(new,p);
}
```

/* DISPOSE di liste */

/* cancella tutti i nodi di una lista, liberando la memoria */

```
void disposelis(node **lis)
{
    node *p;
    while (*lis != NULL)
    {
        p=*lis;
        *lis=(*lis)->next;
        free(p);
    }
}
```

/* cancella il primo nodo di una lista, liberando la memoria */

```
void disposefirst(node **lis)
{
node *p;
    if (*lis != NULL)
    {
        p=*lis;
        *lis=(*lis)->next;
        free(p);
    }
}
```

/* 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(node **lis)
{
node *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 *lis=NULL;
    }
}
```

```
                                        /* duplicazione di una lista */
node *duplist(node *l)
{
  node *p,*head,*tail;
  head=NULL;
  while (l != NULL)
  {
                p=newnode();
                p->data = l->data;
                p->next = NULL;
                if (head == NULL)
                                {head=p; tail=p;}
                else  {tail->next=p; tail=tail->next;}
                l=l->next;
  }
  return head;
}
```

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

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

```
/* date due liste, costruire la lista "somma", posizione per posizione
ad esempio:
l1: 2->4->6->8
l2: 5->4->3
risultato: 7->8->9->8  */
node *buildsumlis(node *l1, node *l2)
{
  node *p, *head, *tail;
  head=NULL;
  while ((l1 != NULL) && (l2 != NULL))
  {
            p=newnode();
            p->data = l1->data+l2->data;
            p->next = NULL;
                if (head == NULL){head=p; tail=p;}
                      else {tail->next=p; tail=tail->next;}
            l1 = l1->next;
            l2 = l2->next;
  }
  if ((l1 == NULL) && (l2 == NULL)) return head;
  else if (l1 == NULL)
                      {tail->next=duplist(l2); return head;}
             else     {tail->next=duplist(l1); return head;}
}

/* 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*/
node *inters(node *l1, node *l2)
{
node *p,*head,*tail;
head=NULL; tail=NULL;
while ((l1 !=NULL) && (l2 != NULL))
  {
   if (l1->data == l2->data)
            {
             p=newnode();
             p->data=l1->data;
             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->data < l2->data) l1=l1->next;
   else l2=l2->next;
  }
return head;
}
```

/* date due liste, restituisce 1 se sono uguali, 0 altrimenti */

```
int equallis(node *l1, node *l2)
{
  int uguali=1;
  while ((l1!=NULL)&&(l2!=NULL)&& uguali)
  {
            if (l1->data != l2->data)
                        uguali=0;
            else
                        {l1=l1->next; l2=l2->next;}
  }
  if (uguali && (l1==NULL) && (l2==NULL))
            return(uguali);
  else      return(0);
}
```

/* 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(node **lis,node **par,node **dis)
{
node *lpar, *ldis, *q;
*par=NULL;
*dis=NULL;
while (*lis != NULL)
  {
  if (((*lis)->data % 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;
  }
}
```

Parte 2: Ricorsione

2.1 Esempi

```
/* 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 */
/* complessita' (si vedano anche NOTE: paragrafo 2.2)
      tempo: O(n)
      spazio: O(n) */

/* si definisca la somma di x ed y supponendo di avere solo a disposizione le operazioni di incremento e decremento ad uno
*/
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' (si vedano anche NOTE: paragrafo 2.2)
      tempo: O(n)
      spazio: O(n) */


/* 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' (si vedano anche NOTE: paragrafo 2.2)
        tempo: O(n)
        spazio: O(n) */
```

2.2 Record di attivazione e complessita' delle funzioni ricorsive

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

Nota: Si ricorda che ogni RA per una funzione contiene celle per
- i parametri
- le variabili locali
- l'indirizzo di ritorno (indicato con IR nelle figure, indica l'istruzione a cui tornare doto la terminazione dell'esecuzione della funzione)

Inoltre, nel caso di funzioni che ritornino un valore diverso da “void”, il RA contiene anche
- una cella per contenere il risultato della funzione. Per convenzione, nel seguito si dara’ come nome a tale cella il nome della funzione stessa.

Si ricorda inoltre che, all'atto della chiamata di una funzione viene allocato sulla stack di esecuzione il suo record di attivazione. Vengono quindi valutati i parametri attuali. Solo a questo punto viene eseguito il corpo della funzione. Terminate le istruzioni nella funzione, il suo RA viene disallocato, e si ritorna nell'ambiente chiamante ad eseguire l'istruzione specificata nell'IR. La terminazione di una funzione involve tuttavia anche la restituzione all'ambiente chiamante del risultato della funzione stessa.

Esaminiamo dapprima l'esecuzione con i RA della versione ricorsiva *non di coda* (si veda la sezione 2.3 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 2-2-1.

(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.

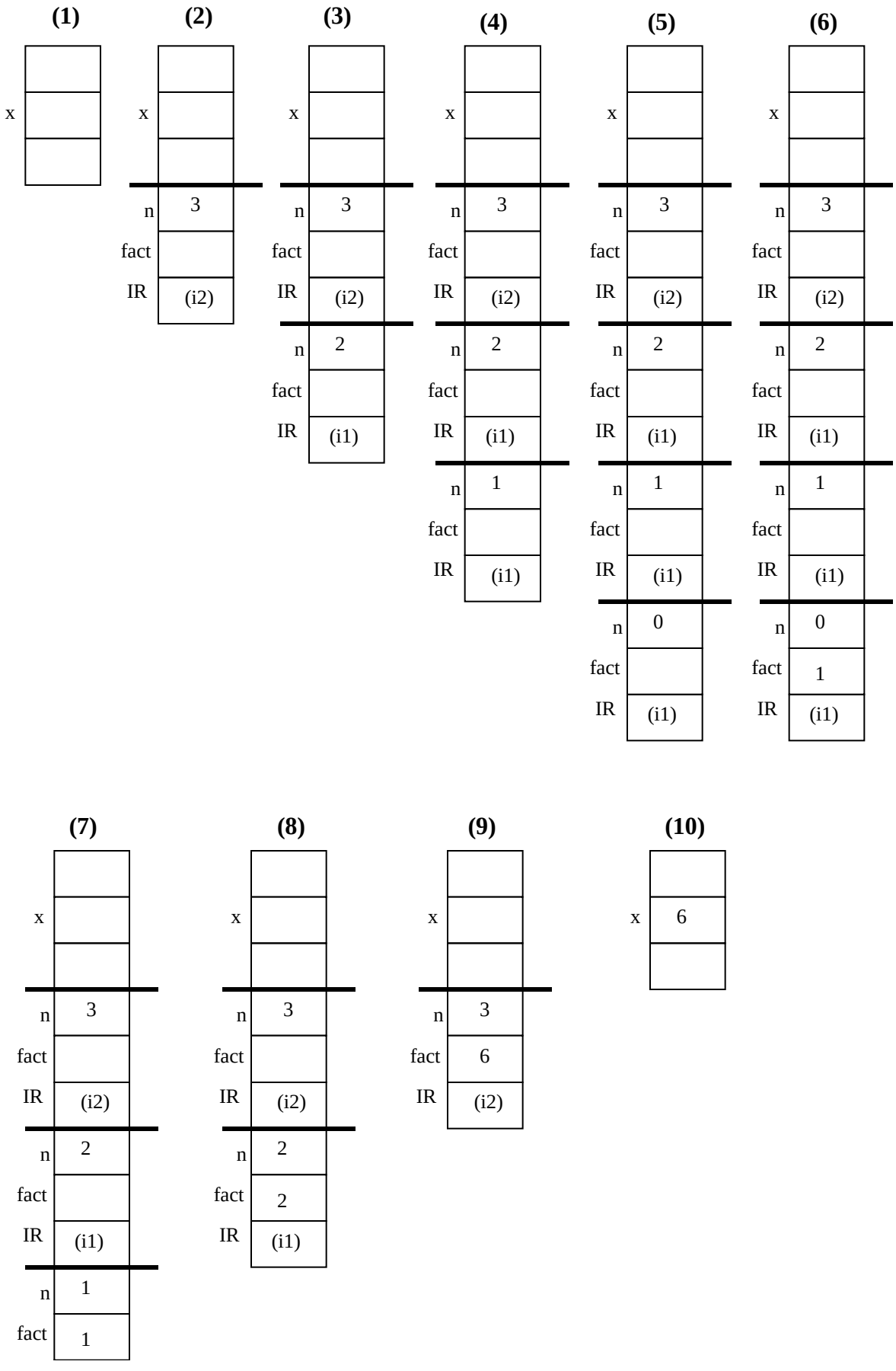
(9) Analogo al precedente, con n=3, fact(2)=2, e quindi il risultato restituito e' 3*2=6.

(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 2-2-1 Fattoriale, ricorsivo non di coda



Consideriamo ora l'esecuzione con i RA della seguente funzione ricorsiva, che calcola il fattoriale con una ricorsione di coda, utilizzando un parametro per riferimento (acc) per accumulare i risultati.

```
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 2-2-2.

(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.

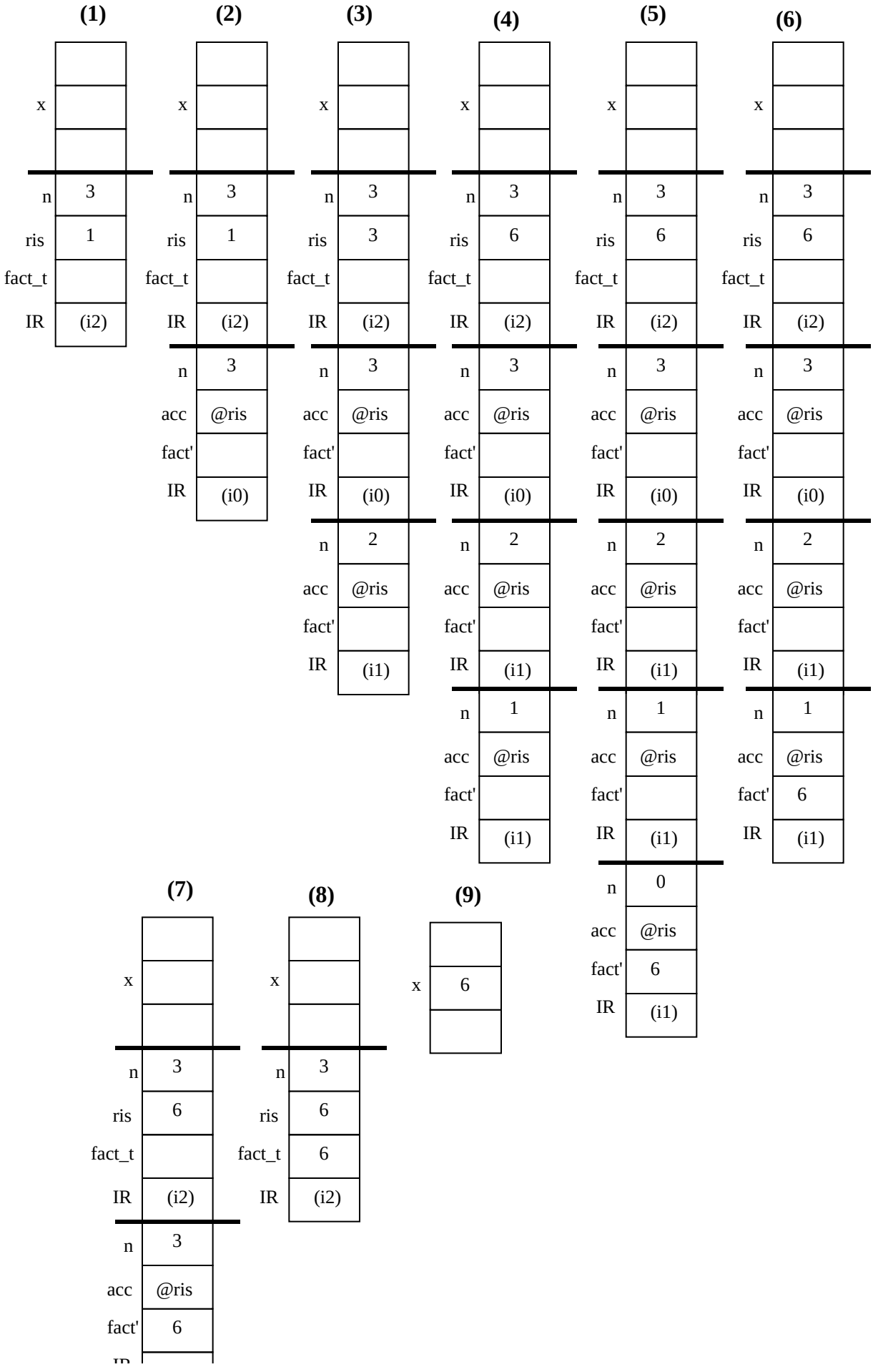
(6) I passi successivi (alcuni passi sono omessi in figura) si limitano a disallocare uno per volta i RA, restituendo sempre al chiamante il valore del risultato computato dalla chiamata ricorsiva. Ad esempio, (6) mostra la situazione dopo che e' disallocato l'ultimo RA, e (7) dopo la disallocazione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

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

Figura 2.2-2 Fattoriale ricorsivo di coda



Proponiamo infine un esempio di esecuzione di una funzione ricorsiva non di coda che genera come risultato una lista. E' interessante notare come, in questo caso, siano "lasciate in sospeso" le operazioni di aggancio dei nodi della lista, che, in ogni ambiente di ricorsione, vengono completate solo dopo aver ottenuto il risultato delle chiamate ricorsive sottostanti.

La funzione nel seguito costruisce una lista di nodi con .data interi positivi leggendo gli interi da tastiera. 0 indica la terminazione della lista. Per i nodi di una lista, si utilizza la definizione di “node” data in precedenza.

```
node *buildlis_rnf(){   int x;    node *p;    printf("nuovo numero da inserire in lista:\n");    scanf("%d", &x);
   if (x<=0) return NULL;
   else
      {
         p=newnode();
         p->data = x;
          p->next=buildlis_rnf();
          return p;
       }
}
```

Nel seguito, si suppone che la funzione buildlis_rnf sia richiamata nel main con l'istruzione lis = buildlis_rnf(); e che l'indirizzo di tale assegnamento sia (i2).
Ad esempio, in figura 2-2-3 e' mostrata l'esecuzione della funzione nel caso in cui l'input letto sia "3 5 0".
Si indichera' con (i2) l'indirizzo dell'assegnamento p->next:=buldlis_rnf; nella funzione buildlis_rnf. Si noti che tale assegnamento puo' avvenire solo dopo la terminazione della chiamata ricorsiva, che restituisce un puntatore ad un nodo.

(1) Il passo (1) mostra la situazione subito dopo il richiamo alla funzione buildlis_rnf nel main. Viene allocato il RA per la funzione, che non contiene celle per i parametri, ma ha due celle, p ed x, per le variabili locali, una (abbreviata con B_L) per il risultato della funzione, ed una per l'IR (che e' i2, ovvero l'assegnamento lis=buildlis_rnf(); nel main). Si noti che sia p che B_L sono celle preposte a contenere dei puntatori a node. In questa situazione si inizia ad eseguire le istruzioni nel corpo della funzione.

(2) scanf("%d", &x) legge 3 dall'input. Poiche' 3>0 si entra nel ramo "else" della funzione. L'esecuzione dell'istruzione p=newnode() genera un nodo di tipo “node”” in uno spazio di memoria esterno al RA, e mette in p un puntatore a tale nodo. P->data=x mette in p->data il valore x. L'assegnamento p->next=buildlis_nrf() e' una chiamata ricorsiva alla funzione. Questa genera l'allocazione di un nuovo RA per la chiamata stessa. Si noti che l'IR del nuovo RA e' l'assegnamento p->next=buillis_rnf(), che sara' eseguito una volta terminata la chiamata ricorsiva.

(3) Come al passo (2).

(4) scanf("%d", &x) legge 0 dall'input. Questo e' il caso base della ricorsione (induzione). Viene quindi eseguito l’istruzione di return NULL. La chiamata alla funzione termina restituendo NULL, ritornando all'istruzione scritta nell'IR, e disallocando il proprio RA.

(5) Nell'ambiente mostrato in (5) viene eseguita l'istruzione
p->next=buildlis_rnf(). Come detto, la chiamata ricorsiva ha restituito in questo caso il valore NULL. Quindi, l'istruzione pone p->next a NULL. Successivamente, l'istruzione “return p” da' come risultato della funzione un puntatore al nodo puntato da p (ovvero, il nodo contenente 5). A questo punto la chiamata ricorsiva termina, e si ritorna all'istruzione scritta nell'IR da eseguire nel RA della funzione chiamante, dopo la disallocazione del RA corrente.

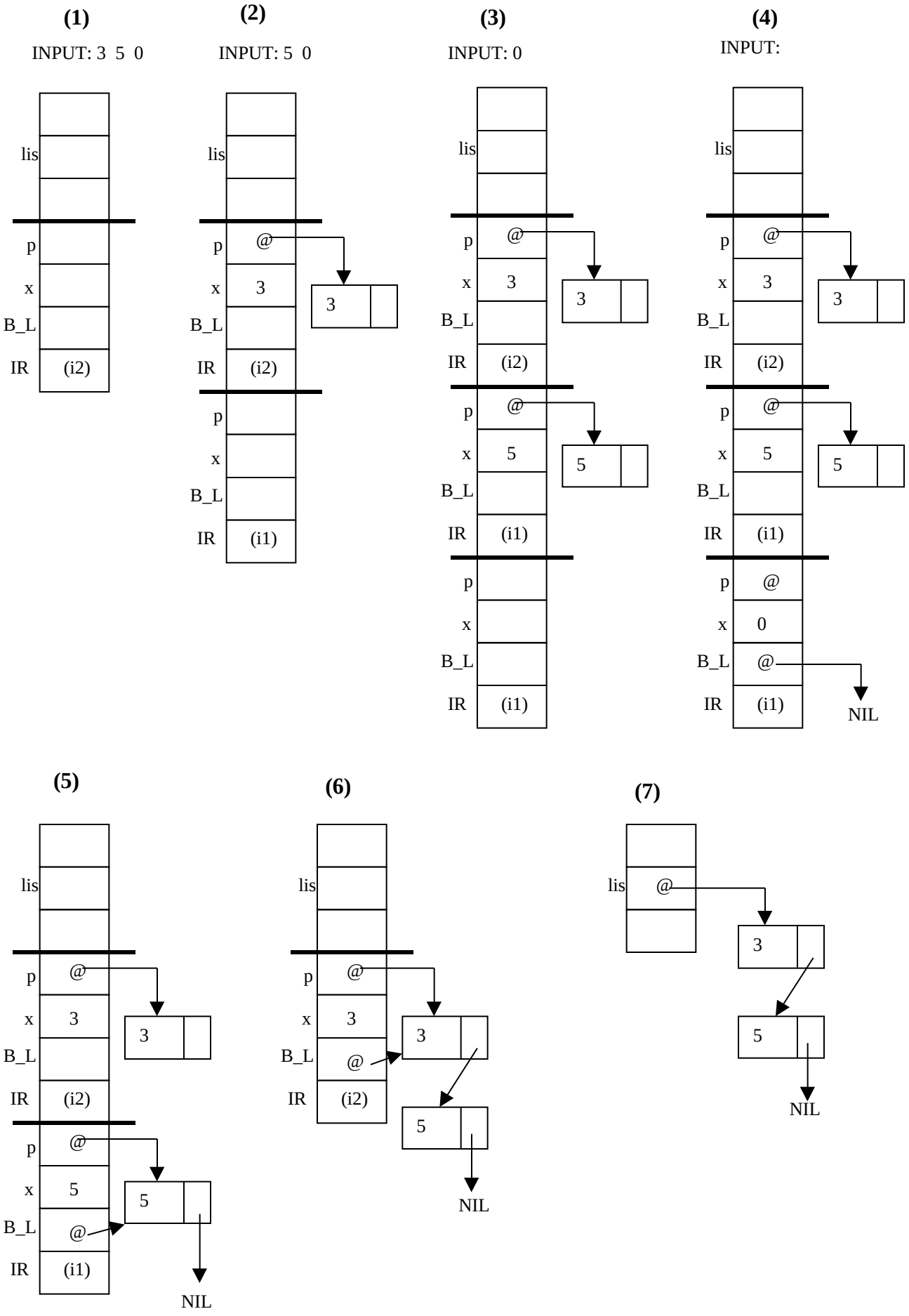
(6) Come al passo (5). Questa volta, il risultato della chiamata riscorsiva e' un puntatore al nodo contenente 5. Quindi l'esecuzione dell'istruzione
p->next=buildlis_rnf() fa si' che il puntatore nel campo next del nodo puntato da p nell'RA corrente (ovvero, del nodo contenente 3) punti al nodo contenente 5, realizzando "l'aggancio" dei nodi nella lista. Successivamente, l'istruzione “return p” da' come risultato della funzione un puntatore al nodo puntato da p (ovvero, il nodo contenente 3). A questo punto la chiamata ricorsiva termina, e si ritorna all'istruzione scritta nell'IR da eseguire nel RA della funzione chiamante, dopo la disallocazione del RA corrente.

(7) Viene finalmente eseguito l'assegnamento lis=buildlis_rnf();, che fa' si' che lis punti alla testa della lista creata.

Nota: anche in questo caso sono state lasciate "in sospeso" delle operazioni, poi completate al ritorno dalle chiamate ricorsive. In particolare, una volta arrivati al passo base (passo 4 in figura 2-2-3), sono gia' stati costruiti i nodi della lista, ma non gli agganci, che vengono realizzati solo in seguito, alla chiusura delle chiamate ricorsive.

Nota2: L'istruzione finale di ritorno “return p” e' essenziale per "restituire all'esterno" il risultato delle varie chiamate ricorsive. Si suggerisce agli studenti di provare a simulare la funzione buildlis_rnf supponendo che tale istruzione non sia presente, al fine di verificare l'importanza di tale istruzione nell'algoritmo dato.

Figura 2.2-3 (buildlis_rnf)



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, la funzione sumA nella sezione 2.1 ha complessita' in tempo O(n), ove n e' la dimensione dell'array (in pratica, ogni chiamata ricorsiva tratta esattamente uno degli elementi dell'array).

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

Nel caso della funzione sumA, come mostrato in precedenza, quando arriviamo al passo base, abbiamo n+1 record di attivazione sulla stack. Anche la complessita' in tempo di sumA e' quindi O(n).

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, la complessita' in tempo della funzione HeapSort di ordinamento e' O(n log n), mentre la complessita' in spazio e' O(log n); si veda il libro di testo.

2.3 Differenti tipi di ricorsione e trasformazioni

E' possibile distinguere fra 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 le funzioni ricorsive riportate nel seguito). 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 fact, sum_inc1 e sumA in 2.1 sono tutte ricorsive lineari, mentre ad esempio la funzione delle torri di Hanoi sul libro di testo anon 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 o funzione ricorsiva e' ricorsiva di coda se contiene al piu' una chiamata ricorsiva (ovvero: se e' ricorsiva lineare) e tale chiamata e' ricorsiva di coda.

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 in 2.1, 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 in 2.1 e' ricorsiva lineare ma non ricorsiva di coda).

Come abbiamo detto, 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 (si riveda l'esecuzione con i record di attivazione di fact_t in 2.2). Per tale motivo, e' possibile trasformare le funzioni ricorsive di coda in funzioni iterative.
In tale ottica, vedremo brevemente le seguenti trasformazioni (cercando di fornirne una definizione il piu' possibile generale):
1) da funzione ricorsiva lineare a ricorsiva di coda
2) da funzione ricorsiva di coda ad iterativa

1) da funzione ricorsiva lineare a ricorsiva di coda
In generale, e' possibile ricondurre molte delle funzioni ricorsive lineari al seguente schema di definizione

(rl1) **f(x)= g(x) se c(x)**
f(x)= h(x,f(k(x)) altrimenti

dove x e' in generale un vettore di valori, c(x) e' il predicato soddisfatto nel caso di base della funzione ricorsiva, e g ed h sono due funzioni che non dipendono da f. Ad esempio, questo schema modella facilmente la definizione di fattoriale, con le seguenti sostituzioni f → fact, c(x) → (x=0), g(x) → 1, h(x,y) → x*y, k(x) → (x-1)

fact(x)= 1 se x=0
fact(x)= x*fact(x-1) altrimenti

In molti casi, lo schema (rl1) puo' essere trasformato nello schema (rc1), che definisce una funzione ricorsiva di coda:

(rc1) **f_coda(x)= f'(x,g(x)) ove**
f'(x,y)= y se c(x)
f'(x,y)= f'(k(x),h(x,y)) altrimenti

Intuitivamente, il passaggio da (rt1) a (rc1) comporta l'introduzione di una nuova variabile che serve da "accumulatore", ovvero per calcolare incrementalmente il risultato della funzione ricorsiva ad ogni passo della ricorsione. Ad esempio, applicando la trasformazione da (rl1) a (rc1) nel caso della funzione fattoriale, otterremo:

fact_coda(x)= fact'(x,1)
fact'(x,y)= y se x=0
fact'(x,y)= fact(x-1,x*y) altrimenti

L'esecuzione con i record di attivazione della versione lineare non di coda e della versione lineare di coda e' stato mostrato in 2.2. E' importante notare come, nel caso della versione ricorsiva di coda, si ottenga il risultato finale nel record di attivazione corrispondente al caso base, ed i record di attivazione sovrastanti nella stack non vengano piu' utilizzati, se non per restituire il risultato al chiamante. In altre parole, mantenere tali record in stack non sarebbe strettamente indispensabile, e cio' giustifica il passaggio da ricorsione di coda ad iterazione (in cui, appunto, solo un record di attivazione e' tenuto sulla stack).

2) da funzione ricorsiva di coda ad iterativa

Dato uno schema di funzione come quello in (rc1), e' in generale possibile ottenere una funzione iterativa corrispondente seguendo il seguente schema:

(it1)

```
…… f"(x)
{
        y= g(x);
        while (!c(x))
        {
                y=h(x,y);
                x=k(x);
        }
        return y;
}
```

Ad esempio, nel caso del fattoriale, si ottiene la seguente funzione:

```
int fact"(int x)
{
        int y=1;
        while (!(x==0))
                {       y= x*y;
                        x= x-1
                }
                return y;
}
```

2.4 Esercizi su Ricorsione

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)

2.4.1 Funzioni ricorsive e ricorsione su vettori

/* 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))*/

```
int somma1(int x, int y)
{
  if (y==0) return x;
  else return 1+somma1(x,y-1);
}
```

/* 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);
}
```

```
/* stampa ricorsiva di un array di n 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 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]);
  }
}

/* somatoria degli elementi di un array (a n elementi) */
/* gli array partono da posizione 0 */
int sumarr_rn(int A[], int n)
{
  if (n<0) return 0;
  else return A[n]+sumarr_rn(A,n-1);
}

/* 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);
  }
}

/* sommatoria degli elementi di un array (a n 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));
}

/* restituisce il minimo di un array di n elementi in un parametro per riferimento */
void minarr_rt(int A[], int n, int *ris)
/* ris inizializzato ad A[0] */
{
  if (n>0)
  {
    *ris= min(*ris,A[n]);
    minarr_rt(A,n-1,ris);
  }
}
```

```
/* 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);
}
```

2.4.2 Ricorsione su liste

```
/* stampa tutti gli elementi di una lista */
/* la funzione riceve in input un puntatore lis al primo nodo di una lista;
   se la lista e' vuota, lis==NULL */
void printlis_rt(node *lis){if (lis != NULL)  {  printf(">>>> %d\n", lis->data);  printlis_rt(lis->next);
}}
/* passo base: se la lista e' vuota (ovvero: lis==NULL), non c'e' nulla da fare */
/* ipotesi induttiva: poiche' (se lis != NULL) la lista che inizia in lis->next ha un nodo in meno di quella che inizia in lis, e' corretto fare l'ipotesi induttiva che printlis_rt(lis->next) stampi tutti i nodi della sottolista
da lis->next fino alla fine della lista */
/* passo induttivo: la stampa di tutti i nodi da lis fino alla fine della lista si ottiene stampando lis->data e poi il resto della lista */
/* terminazione: ad ogni richiamo, lis si sposta al nodo successivo della lista (ovvero: a lis->next), fino a quando non si raggiunge il passo base (lis==NULL, ovvero: fino a quando non si raggiunge la fine della lista */
/* complessita' (si vedano anche NOTE: paragrafo 2.2)
        tempo: O(n)
        spazio: O(n)
        ove n e' il numero di nodi della lista */
```

Nota: Nel seguito, per brevita', passo base, ipotesi induttiva e passo induttivo verranno omessi nella descrizione delle funzioni

Costruzione ricorsiva di una lista. Dato un intero n, le funzioni nel seguito costruiscono una lista di nodi contenenti gli interi da 1 ad n. Le varie funzioni differiscono nel modo di utilizzare le chiamate ricorsive ed i parametri. Si consigliano gli studenti di eseguire e confrontare le varie funzioni di costruzione con i recoord di attivazione.
La complessita' in spazio e tempo e' sempre O(n).

/* se si vogliono generare i nodi nell'ordine da 1 ad n (e mantenerli in ordine)si usa una ricorsione non di coda */**node* buildlis_n_rnf (int n, int cur)/* cur deve essere inizializzato ad 1 */**
{ node *p; if (cur > n) return NULL; else { p=newnode(); p->data=cur; p->next=buildlis_n_rnf(n,cur+1); return p; }}

/* se si vogliono generare i nodi nell'ordine (e mantenerli in ordine)
si usa una ricorsione non di coda anche se si usa un parametro
per contenere il risultato */
void buildlis_n_rn (int n, int cur, node **ris)
/* cur deve essere inizializzato ad 1 */
/* occorre un puntatore a puntatore, e ris deve essere
inizializzato a NULL */
```
{
    node *p,*q;
    if (cur > n) *ris=NULL;
    else
    {
        p=newnode();
        p->data=cur;
        buildlis_n_rn(n,cur+1,&q);
      p->next=q;
      *ris=p;
     }
}
```

/* se NON vogliono generare i nodi nell'ordine (ma si vuole lista crescente)si puo' usare una inserzione in testa, e non lasciare agganci in sospeso */
void buildlis_n_rt (int n, node **ris)
/* occorre un puntatore a puntatore, e ris deve essere
inizializzato a NULL */
```
{
    node *p;
    if (n > 0)
            {
     p=newnode();
     p->data=n;
      p->next=*ris;
      *ris=p;
     buildlis_n_rt(n-1,ris);
     }
}
```

Costruzione ricorsiva di una lista. La funzione nel seguito costruisce una lista di nodi con .data interi positivi leggendo gli interi da tastiera. 0 indica la terminazione della lista.

```
/* costruzione ricorsiva di una lista leggendo i numeri da input */node *buildlis_rnf(){     int x;     node *p;
printf("nuovo numero da inserire in lista:\n");     scanf("%d", &x);
    if (x<=0) return NULL;
    else
       {
              p=newnode();
              p->data = x;
              p->next=buildlis_rnf();
              return p;
       }
}
```

Complessita' in spazio e tempo: O(n), ove n e' il numero di interi >0 letti

Nota: la maggior parte delle funzioni nel seguito presuppone di ricevere in input una (o piu') lista costruta prima del richiamo della funzione stessa.

Visita di una lista

/* VISITA CON CONTATORE E CONDIZIONE */

/* stampa i numeri della lista in posizione multipla di x */**void printpos_rt(node *lis, int x, int pos)**/* pos inizializzato ad 1 */**{if (lis != NULL) { if ((pos % x)==0) printf(">>>> %d\n", lis->data); printpos_rt(lis->next,x,pos+1); }}**

/* RICERCA DI UN ELEMENTO IN UNA LISTA */

/* restituisce un puntatore all'elemento n-esimo di una lista */**node *nth_rt(int n, node *p)**/* n => 0 */**{ if (p == NULL) return NULL; else if (n==1) return p; else return nth_rt(n-1,p->next);}**

/* ricerca di un elemento x nella lista p*/**node *find_rt(int x, node *p){ if (p == NULL) return NULL; else if (p->data==x) return p; else return find_rt(x, p->next);}**

/* conta le occorrenze di x in una lista */**int countx_rn(int x, node *lis){if (lis == NULL) return 0;else if (lis->data == x) return 1+countx_rn(x, lis->next); else return countx_rn(x, lis->next);**

}

Operatori aggregati su lista

/* sommatoria (non condizionata) di tutti gli elementi in una lista. Versione non ricorsiva di coda (tail) */
```
int sumlis_rn(node *lis){ if (lis == NULL) return 0; else return lis->data+sumlis_rn(lis->next);
}
```

/* sommatoria (non condizionata) di tutti gli elementi in una lista. Versione ricorsiva di coda */
```
void sumlis_rt(node *lis, int *ris){ if (lis != NULL) { (*ris)+=lis->data;
sumlis_rt(lis->next,ris); }}
```

/* MODIFICA DI UNA LISTA: OPERAZIONI SUI LINK */

```
/* crea un nuovo nodo contenente x e lo inserisce dopo
la prima occorrenza di y */
void insaftx_rt(node *lis, int y, int x)
{
    if (lis != NULL)
            if (lis->data == y)
            {
                    node *p;
                    p=newnode();
                    p->data=x;
                    p->next=lis->next;
                    lis->next=p;
            }
      else insaftx_rt(lis->next,y,x);
}

/* crea un nuovo nodo contenente x e lo inserisce dopo  OGNI occorrenza di y */void insaftallx_rt(node *lis, int y, int x){
if (lis != NULL)                if (lis->data == y)
            {
                    node *p;
                    p=newnode();
                    p->data=x;
                    p->next=lis->next;
                    lis->next=p;
/* NB lis->next->next perche' ho inserito un nuovo nodo, che devo saltare -se no potrei avere loop per x=y */
                insaftallx_rt(lis->next->next,y,x);
       }
    else insaftallx_rt(lis->next,y,x);
}
```

/* CANCELLAZIONE */

/*cancellazione di tutte le occorrenze di x da una lista -- versione a 2 ricorsioni*/

/* elimina le x dalla testa della lista */**void delfirsts_rt(int x, node **lis){if (*lis != NULL) if ((*lis)->data==x)**
{
***lis=(*lis)->next;**
delfirsts_rt(x, lis);
}
}
/* elimina le x da dentro la lista */**void delinx_rt(int x, node *lis){node *p;if (lis->next != NULL) if (lis->next->data == x) { p=lis->next; lis->next=lis->next->next;**
free(p);
delinx_rt(x,lis);
}
else delinx_rt(x,lis->next);
}
/* funzione generale */**void delallx_rt(int x, node **lis){ delfirsts_rt(x,lis); delinx_rt(x,*lis);}**

Copie di liste

```
/* copia di lista ricorsiva non tail */node *copy_rnf(node *l){    node *p;            if (l == NULL) return NULL;       else
     {
              p=newnode();
         p->data = l->data;
         p->next = copy_rnf(l->next);
              return p;
     }
}

/* copia di lista ricorsiva tail (con inversione) *//* NB ottengo la lista INVERTITA!! */node *copy_rt(node *l, node
**acc)
/* acc inizializzato a NULL */
{
  node *p;
  if (l != NULL)
     {
        p=newnode();
         p->data = l->data;
         p->next = (*acc);
        (*acc)=p;
        return copy_rt(l->next,acc);
     }
}

/* copia di lista ricorsiva tail */
/* NB ottengo la listan NON INVERTITA!! */
void copy_rt2(node *l, node **head, node **tail)
/* *head, *tail inizializzati a NULL */
{
  node *p;
  if (l != NULL)
     {
              p=newnode();
          p->data = l->data;
          p->next = NULL;
              if ((*head)==NULL)
                        {(*head)=p; (*tail)=p;}
              else      {(*tail)->next=p; (*tail)=(*tail)->next;}          copy_rt2(l->next,head,tail);       }}
```

/* copia condizionata di lista (es: duplica solo I nodi maggiori di x) */

```
node *duplistcond_rnf(node *l, int x)
{
  node *p;
  if (l == NULL) return NULL;
      if (l->data > x)
       {
                p=newnode();
                p->data = l->data;
                p->next = duplistcond_rnf(l->next,x);
                return p;
       }
  else return duplistcond_rnf(l->next,x);
}
```

/* date tre liste l1, l2 ed l3, dare 1 se l3 e' la "somma" di l1 ed l2 -posizione per posizione- 0 altrimenti */
/* Ad esempio, dato il seguente input, la funzione restituisce 1
l1: 3→5→9→2→5→6
l2: 8→5→7→8
l3: 11→10→16→10→5→6 */

```
int issum(node *l1, node *l2, node *l3)
{
  if ((l3==NULL)&&(l2==NULL)&&(l1==NULL)) return 1;
  else if (l3==NULL) return 0;
  else if ((l1==NULL)&&(l2==NULL)) return 0;
  else if ((l1!=NULL)&&(l2==NULL))
       return ((l1->data==l3->data) &&
                    issum(l1->next,l2,l3->next));
  else if ((l1==NULL)&&(l2!=NULL))
       return ((l2->data==l3->data) &&
                    issum(l1,l2->next,l3->next));
  else
       return (((l1->data+l2->data)==l3->data) &&
                    issum(l1->next,l2->next,l3->next));
}
```

/* date due liste ordinate e senza duplicati, generare la lista unione (senza duplicati), SENZA generare nuovi nodi */

```
node *unione(node *l1, node *l2)
{
  node *p;
  if ((l1 !=NULL) && (l2 != NULL)) return NULL;
  else if (l1==NULL) return l2;
  else if (l2==NULL) return l1;
  else if (l1->data==l2->data)
            {
             l1->next=unione(l1->next,l2->next);
             return l1;
            }
  else if (l1->data<l2->data)
            {
             l1->next=unione(l1->next,l2);
             return l1;
            }
  else
            {
             l2->next=unione(l1,l2->next);
             return l2;
            }
}
```

/* data in input una lista lis ed un intero x, restituisce in output due liste: minu, che contiene tutti i nodi minori uguali di x, e mag, con i nodi maggiori di x, senza creazione di nodi */

```
void splitmag(node **lis,node **minu, node **lminu, node **mag, node **lmag, int x)
/* *minu, *lminu, *mag, *lmag tutti inizializzati a NULL
minu e' un puntatore al puntatore in testa ai minori o uguali,e lminu il puntatore al puntatore alla coda */
{
if (*lis == NULL)
    {
            if (*lminu != NULL) (*lminu)->next=NULL;
            else {*lminu=NULL;*minu=NULL;}
            if (*lmag != NULL) (*lmag)->next=NULL;
            else {*lmag=NULL;*mag=NULL;}
    }
   else
    {
            if ((*lis)->data <= x)
                    if (*minu == NULL)
                            {*minu=*lis; *lminu=*lis;}
                    else    {(*lminu)->next=*lis; *lminu=*lis;}
            else
                    if (*mag == NULL)
                            {*mag=*lis; *lmag=*lis;}
                    else    {(*lmag)->next=*lis; *lmag=*lis;}
            splitmag(&((*lis)->next),minu,lminu,mag,lmag,x);
/* nota: devo passare un puntatore a (*lis)->next */
    }
}
```