L06. RICORSIONE

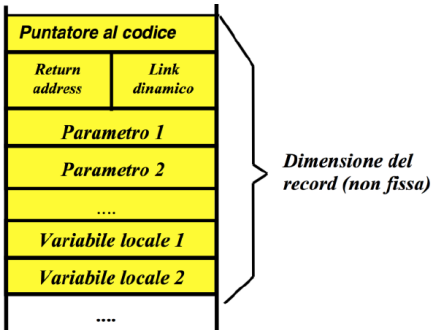
Tecnica usata in programmazione per la quale un sotto-programma chiama se stesso.

RECORD DI ATTIVAZIONE:

Il sistema operativo gestisce la ricorsione tramite una zona di memoria chiamata Stack, in cui vengono inseriti all'interno di questo Stack dei **record di attivazione**, quest'ultima è una struttura dati creata dinamicamente ogni volta che viene invocata una funzione, in particolare, ciò che avviene nel nostro sistema è che:

1. si crea una nuova **attivazione** (o istanza) della funzione chiamata;
2. viene **allocata la memoria** per i parametri e le variabili locali;
3. si effettua il **passaggio dei parametri**;
4. si **trasferisce il controllo** alla funzione chiamata;
5. si **esegue il codice** della funzione.

Struttura record di attivazione:



Questi sono i campi che troviamo all'interno del record di attivazione, in particolare, troviamo:

- i **parametri formali**, cioè la copia dei parametri che vengono passati alla nostra funzione;
- le **variabili locali**;
- degli indirizzi che servono a gestire il flusso di istruzioni del programma in particolare abbiamo un **indirizzo di ritorno** che indica il punto a cui tornare (nel codice del chiamante) al termine della funzione;
- un collegamento al record di attivazione del chiamante, detto **link dinamico**;
- l'**indirizzo del codice** della funzione, cioè un puntatore alla prima istruzione del corpo di questa funzione, cioè quella a cui è riferito questo record di attivazione.

Per quanto riguarda il **ciclo di vita** di un record di attivazione, cioè quando viene creato e quando viene distrutto.

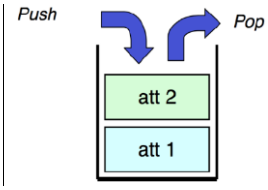
Un nuovo record di attivazione viene creato nel momento in cui si invoca una funzione *f* e permane per tutto il tempo in cui questa funzione è in esecuzione e viene invece distrutto al termine dell'esecuzione della funzione.

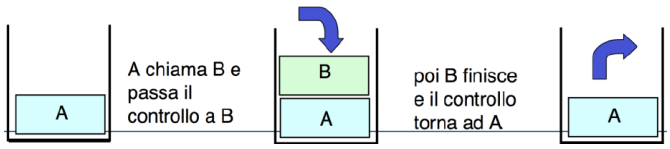
È chiaro che per ogni invocazione di una stessa funzione ci sarà un nuovo record di attivazione, quindi il record di attivazione non è associato alla funzione ma associato alla sua esecuzione, quindi se eseguiremo all'interno di un programma n volte una funzione *f* avremo n volte n record di attivazioni corrispondenti a quella funzione.

NOTA: funzioni diverse potrebbero avere una dimensione del record di attivazione differente, invece, la stessa funzione ogni volta che viene invocata, ha sempre la stessa dimensione del record e si può conoscere a priori.

Il sistema operativo, per l'esecuzione di un programma, mantiene un'area di memoria, in cui vengono allocati record di attivazione, che viene gestita come una lista **LIFO** e corrisponde esattamente ad uno **Stack**, ogni elemento di questo Stack è appunto un record di attivazione.

Come già visto, lo Stack supporta operazioni di **push** e **pop**, la prima ci consente di aggiungere un elemento in cima allo Stack e la seconda invece di prelevarlo ed eliminarlo dalla cima dello Stack, anche questa struttura del sistema operativo che mantiene i record di attivazione, funzione esattamente in questo modo.





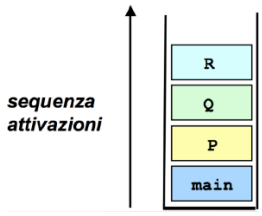
Supponiamo di avere una funzione A che chiama una funzione B, nello Stack abbiamo soltanto il record di attivazione della funzione A, poi questa chiama B e passa il controllo a B, quindi quello che viene fatto è una Push all'interno dello Stack in cui viene inserito il record di attivazione corrispondente all'esecuzione della funzione B, poi B finisce il controllo e quindi il controllo torna ad A e quello che abbiamo è una pop di B dallo Stack, quindi nel nostro Stack sarà scomparso perché viene distrutto il record di attivazione corrispondente a B.

Supponiamo di avere 4 funzioni che si chiamano tra loro, in particolare:

- `int R(int a) { return a+1; }`
- `int Q(int x) { return R(x); }`
- `int P(void ) { int a=10; return Q(a); }`
- `main() { int x = P(); }`

Avremo una sequenza di chiamate che effettuerà il sistema operativo:

S.O. → main → P() → Q() → R()



RICORSIONE:

Fatta quindi questa premessa su come il sistema operativo gestisce le chiamate a funzione tramite questa struttura, vediamo adesso che cos'è la ricorsione e che cosa comporta una chiamata ad una funzione ricorsiva o un sottoprogramma ricorsivo.

Un **sottoprogramma ricorsivo** è un sottoprogramma che direttamente o indirettamente, cioè tramite un'altra funzione, chiama sé stesso. I linguaggi che possono gestire la ricorsione lo fanno mediante la gestione dei **record di attivazione**.

Prima di scrivere una funzione ricorsiva dobbiamo pensare ad un approccio che risolva ricorsivamente un problema, questo approccio prevede:

- L'**identificazione di un caso base**, in cui c'è una soluzione nota;
- Esprimere la soluzione al **caso generico n in termini dello stesso problema** in uno o più casi più semplici, per casi più semplici si intende la risoluzione dello stesso problema ma con una taglia inferiore dell'input, quindi se la funzione risolve il problema per un generico caso n le funzioni che verranno chiamate ricorsivamente risolvono il problema per n-1, n-2 o cose simili.

Molte **funzioni matematiche** sono definite ricorsivamente quando nella sua definizione compare un riferimento a sé stessa.

La dimostrazione che queste funzioni risolvono il problema è basata sul principio di **induzione matematica**:

- se una proprietà P vale per $n=n_0$ (**CASO BASE**);
- si può provare che, **assumendola valida per n**, allora vale per n+1 (allora P vale per ogni $n \geq n_0$).

La funzione fattoriale(n), denotato come n!, è definito per tutti gli interi n≥0 come:

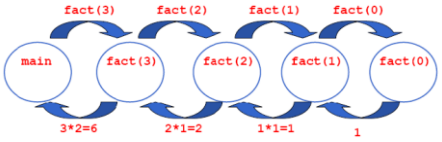
- n! = 1 se n=0;
- n! = n*(n-1)! Se n>0.

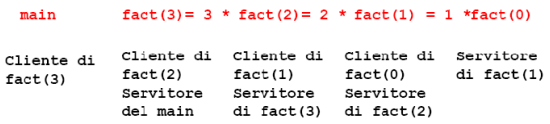
Data una funzione ricorsiva, questa fa sia da **Servitore** (funzione chiamata) che **Cliente** (funzione chiamante):

```
Int fact(int n){
  if(n==0) return 1;
    else return n*fact(n-1);
}
main() {
  int fz, z= 5;
  fz= fact(z-2);
}
```

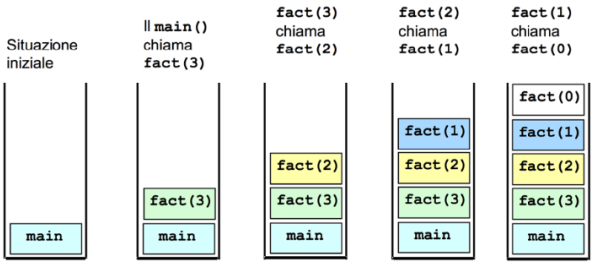
1. Il main chiama fact(3);
2. fact(3) chiama fact(2);
3. fact(2) chiama fact(1);
4. fact(1) chiama fact(0);
5. fact(0) restituisce 1;
6. fact(1) restituisce 1;
7. fact(2) restituisce 2;
8. fact(3) restituisce 6;
9. il main riceve 6 da fact(3) e lo assegna alla variabile fz.

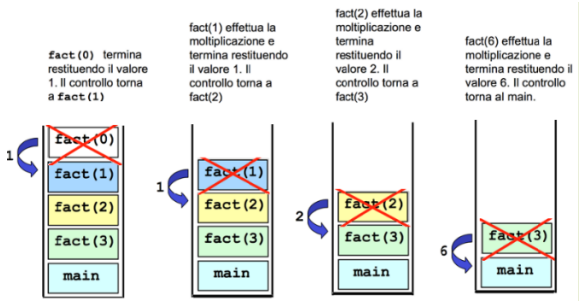
La si può rappresentare anche:





Nello Stack di sistema avviene la seguente situazione:

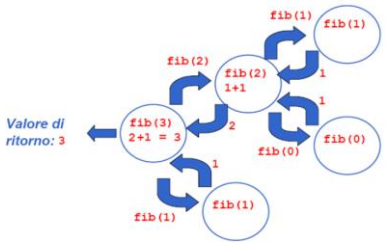




Un altro esempio famoso di funzione ricorsiva è la **successione di Fibonacci**, detta anche successione Aurea, è una successione di interi positivi in cui ciascun numero è la somma dei due precedenti, i primi due invece sono per definizione pari a 1.

Definizione ricorsiva di questa successione:

- La base è costituita da due relazioni: $F_0 = 1$ ed $F_1 = 1$;
- L'ennesimo numero di Fibonacci: $F_n = F_n - 1 + F_n - 2$ per ogni $n > 1$.



Negli esempi visti finora, si inizia a sintetizzare il risultato «**a ritroso**», solo dopo che le chiamate si sono chiuse.

Il risultato viene sintetizzato a partire dalla fine, perché occorre prima arrivare al caso «**banale**»:

- Il caso banale fornisce il valore di partenza;
- Poi si sintetizzano a ritroso i successivi risultati parziali.

ITERAZIONE VS RICORSIONE:

Vediamo le caratteristiche principali che distinguono la ricorsione dall'iterazione. Prendiamo in considerazione un una funzione che calcola il fattoriale in modo iterativo:

```
int fact(int n){
      int i=1;
      int F=1; /*inizializzazione del fattoriale*/
      while (i <= n)
            {F=F*i;
             i=i+1; }
      return F;
      }
```

DIFFERENZA CON LA VERSIONE RICORSIVA: ad ogni passo viene accumulato un risultato intermedio

La variabile F accumula risultati intermedi: se n = 3 inizialmente F=1, poi al primo ciclo F=1, poi al secondo ciclo F assume il valore 2. I ne all'ultimo ciclo i=3 e F assume il valore 6

- Al primo passo F accumula il fattoriale di 1
- Al secondo passo F accumula il fattoriale di 2
- Al passo i-esimo F accumula il fattoriale di i

21

La caratteristica principale dell'iterazione è quella che per calcolare un risultato, i risultati intermedi vengono portati **in avanti**, possiamo dire che il risultato viene sintetizzato in avanti e ad ogni passo è disponibile quindi un risultato parziale, dopo K passi si ha a disposizione un risultato parziale relativo al caso K.

Questo invece non avviene nei processi computazionali ricorsivi in cui non abbiamo alcun risultato finché non giungiamo al caso elementare, cioè alla base della ricorsione.

In realtà è possibile simulare un processo computazionale iterativo attraverso funzioni ricorsive, in questo caso per simulare l'iterazione utilizziamo una variabile che chiamiamo **accumulatore** e praticamente questa variabile è destinata ad esprimere in ogni istante la soluzione corrente e viene passato come parametro ad ogni chiamata della funzione. Questo tipo di ricorsione che simula un processo computazionale iterativo è detta **ricorsione Tail**.

RICORSIONE TAIL:

Questa ricorsione, che realizza un processo computazionale iterativo, è detta anche "**ricorsione apparente**", perché è una ricorsione che in realtà simula un'iterazione.

La chiamata ricorsiva, in queste funzioni che utilizzano la ricorsione apparente, è sempre l'ultima istruzione, questa chiamata serve soltanto per proseguire la computazione, i calcoli invece vengono fatti prima, da questo il nome **ricorsione Tail**.

È possibile ***trasformare*** una funzione iterativa in una funzione ricorsiva che effettua la stessa computazione. Prendiamo il ciclo iterativo e lo trasformiamo in un if con la stessa condizione, il corpo del ciclo rimane invariato, quindi copiamo ciò che abbiamo, ricordandoci però che come ultima istruzione dell'if metteremo la chiamata ricorsiva per far continuare la computazione.

`while (condizione) {`
`  <corpo del ciclo>`
`}`
	`if (condizione) {`
	`  <corpo del ciclo>`
	`  <chiamata ricorsiva>`
	`}`

È chiaro che la funzione ricorsiva dovrà ospitare dei nuovi parametri per portare avanti le variabili di stato.

Vediamo la funzione fattoriale nella versione ricorsione Tail:

```
int fact(int n){
  return factIt(n,1,1);
}

int factIt(int n, int F, int i){
  if (i <= n)
   {F = i*F;
    i = i+1;
    return factIt(n,F,i);
   }
  return F;
}
```

Inizializzazione dell'accumulatore: corrisponde al fattoriale di 1

Contatore del passo

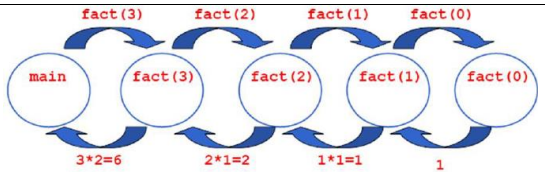
Accumulatore del risultato parziale

Al posto del l'iterazione che avevamo prima con while mettiamo una condizione con un if, la condizione all'interno rimane la stessa e anche le prime due istruzioni che avevamo nel while rimangono le stesse, quindi abbiamo la moltiplicazione di i*F riassegnata ad F e l'incremento della variabile i e poi infine aggiungiamo come ultima istruzione all'interno del blocco del if la chiamata ricorsiva.

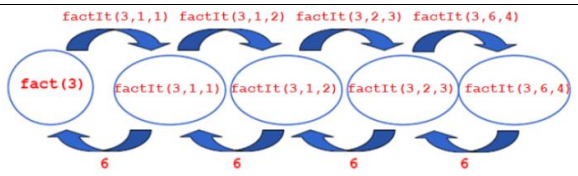
La funzione non ha più un solo parametro, cioè N (numero del quale intendiamo calcolare il fattoriale), ma abbiamo aggiunto due parametri che portiamo avanti per continuare la computazione, F che è l'accumulatore (variabile che conserva il risultato parziale) e la i che è il contatore del passo che ci dice in quale passo siamo arrivati.

Chiamiamo questa funzione factIt all'interno di una funzione fact che prende soltanto n come parametro, questa funzione serve per nascondere la chiamata alla ricorsione Tail e questa prima chiamata che facciamo prendere i parametri inizializzati correttamente.

RICORSIONE CLASSICA



RICORSIONE TAIL



Quindi questa soluzione ricorsiva per il fattoriale ha tutte le caratteristiche di una funzione ricorsiva poiché esegue una computazione ricorsiva di fatto, ma in realtà dà luogo ad un processo computazionale iterativo per questo motivo la chiamiamo ricorsione apparente o ricorsione Tail.

La differenza principale con la ricorsione classica è questa caratteristica che il risultato viene sintetizzato in avanti, quindi il calcolo avviene nel corpo del if e poi viene portato avanti. Arrivati alla fine non si fa altro che riportare indietro fino al chiamante il risultato già ottenuto.

STRUTTURE RICORSIVE:

Alcune strutture dati sono inerentemente ricorsive, questo vale per molte delle sequenze e delle strutture ad albero. Su questo tipo di strutture la formulazione ricorsiva di algoritmi risulta essere molto naturale.

Ad esempio, una lista può essere definita come una lista vuota oppure un elemento seguito da una lista, il primo caso somiglia alla base di una ricorsione, mentre il secondo invece somiglia ad un passo ricorsivo, quindi a tutti gli effetti una lista è una struttura ricorsiva si può fare lo stesso ragionamento anche per gli alberi. Possiamo implementare alcuni algoritmi per la **lista linkata** in modo ricorsivo:

- **Stampa lista**: stampo l'elemento corrente p e chiamo ricorsivamente la funzione di stampa sulla lista puntata da p->next;
- **Ricerca**: verifico se il dato cercato è presente nell'elemento corrente (in caso affermativo restituisco l'elemento) e chiamo ricorsivamente la funzione di ricerca sulla lista puntata da p->next;
- **Numero di occorrenze di un item**: verifico se il dato cercato è presente nell'elemento corrente (in caso affermativo incremento il contatore) e chiamo ricorsivamente la funzione di conteggio sulla lista puntata da p->next;
- **Deallocazione degli elementi**: chiamo ricorsivamente la funzione di deallocazione sulla lista puntata da p->next e libero la memoria corrispondente all'elemento corrente p.

Le differenze tra la ricorsione e l'iterazione, entrambe effettuano una computazione che ripete le stesse istruzioni, nell'interazione questa viene fatta tramite la dichiarazione di un ciclo, nella ricorsione invece la computazione viene ripetuta attraverso chiamate alla stessa funzione. Per terminare la computazione l'iterazione utilizza la condizione all'interno del ciclo iterativo che fallisce quando si termina, invece nella ricorsione si arriva al caso base e poi si va a ritroso. Entrambe sia la ricorsione che l'iterazione, possono dar luogo a cicli infiniti quindi andare in loop.

Una prima caratteristica a favore dell'iterazione è quella relativa alla performance, infatti la ricorsione richiede un notevole overhead o sovraccarico al tempo di esecuzione dovuto alla gestione dello Stack, ogni chiamata a funzione presuppone la creazione di un record di attivazione e dell'inserimento di questo all'interno dello Stack, di contro invece la ricorsione è una buona pratica di Ingegneria del software perché risulta spesso essere più chiara come lettura del codice.

Consigliamo quindi l'utilizzo di algoritmi ricorsivi quando dobbiamo implementare delle funzioni che sono effettivamente ricorsive, abbiamo ad esempio visto fattoriale e Fibonacci che per la loro stessa definizione conviene che vengano implementate ricorsivamente, abbiamo anche detto che molte strutture presentano una natura inerentemente ricorsiva quindi su queste strutture, come strutture ad albero o sequenze, spesso la formulazione ricorsiva di algoritmi risulta essere più naturale, la ricorsione invece andrebbe evitata quando la soluzione iterativa è abbastanza ovvia e quando le prestazioni sono un elemento critico del nostro programma.

```
void recursivePrintList(struct node *p){
   if(p != NULL) {
      outputItem(p->item);
      recursivePrintList(p->next);
   }
}

void printList(List list){
  recursivePrintList(list->head);
  printf("\n");
}
```

```
struct node * minimo (struct node *p){
   struct node *i, *min = p;
   for (i = p; i != NULL; i = i->next){
      if ((cmpItem(min->item, i->item)) > 0)
         min = i;
   }
   return min;
}

void recursiveSort(struct node *p){
   if(p != NULL) {
      struct node *pos_minimo = minimo(p);
      swap(&(pos_minimo->item), &(p->item));
      recursiveSort(p->next);
   }
}
void sortList(List list){
  recursiveSort(list->head);
}
```

```
Item recursiveSearchList(struct node *p, Item item, int *pos){
   if(p != NULL) {
      if(cmpItem(item,p->item) == 0)
         return cloneItem(p->item);
      else {
         ++*pos;
         return recursiveSearchList(p->next, item, pos);
      }
   } else {
      return NULL;
   }
}
Item searchList(List list, Item item, int *pos){
   *pos=0;
   struct node *result = recursiveSearchList(list->head, item, pos);
   if(result == NULL)
      *pos=-1;
   return result;
}
```