RICORSIONE

Forma di programmazione alternativa all'iterazione

Funzione/procedura ricorsiva → Una funzione si dice ricorsiva se richiama se stessa

```
int fatt (int n){
    if (n<=1)
        return 1;
    else
        return n * fatt(n-1);
}
```

informatico

non è una chiamata in coda

$$\begin{cases} fatt(1) = fatt(0) = 1 \\ fatt(n) = n * fatt(n-1) \end{cases}$$

matematico

$fie(1) = fie(1)$ NO Induttivo

ricorsivo

```
int fie (int n){
    if (n=1)
        return fie(1)
}
```

RICORSIONE

Forma di programmazione alternativa all'iterazione

Funzione/procedura ricorsiva → Una funzione si dice ricorsiva se richiama se stessa

```
int fatt (int n){
    if (n<=1)
        return 1;
    else
        return n * fatt(n-1);
}
```

informatico

non è una chiamata in coda

$$\begin{cases} fatt(1) = fatt(0) = 1 \\ fatt(n) = n * fatt(n-1) \end{cases}$$

matematico

$fie(1) = fie(1)$ NO Induttivo

ricorsivo

```
int fie (int n){
    if (n=1)
        return fie(1)
}
```

$$\begin{cases} foo(0) = 0 \\ foo(n) = foo(n+1) \end{cases}$$

no induttivo

```
int foo (int n) {
  if (n=0) return 0;
  else return foo(n+1)
}
```

ricorsivo

Perché un linguaggio permette la ricorsione

↳ Procedure nel linguaggio

↳ Gestione dinamica della memoria

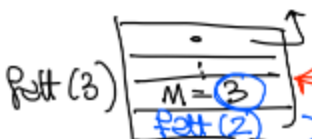
Ricorsione in coda → forma ottimizzata (rispetto all'uso della memoria) della ricorsione

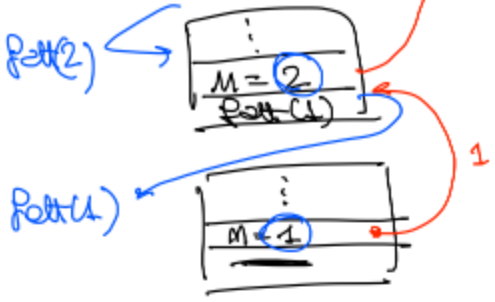
Una chiamata (dentro f) ad una procedura (g) si dice in coda se f restituisce il valore richiesto da g senza ulteriori computazioni

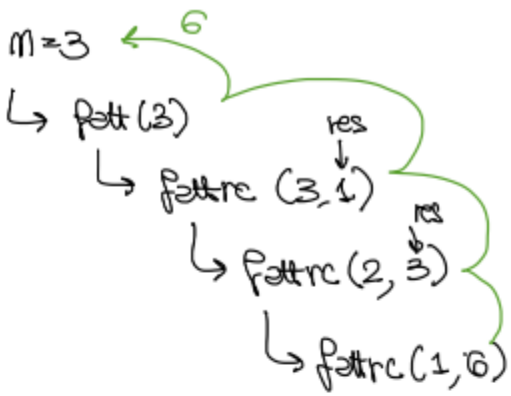
Una procedura si dice ricorsiva in coda se la chiamata ricorsiva è in coda

```
int fatt (n) {
    return fattrc (n, 1);
}

int fattrc (int n, int res) {
  if (n<=1) return res;
  else return
      fattrc (n-1, res * n)
```







$Fib(0) = 1$

$Fib(1) = 1$

$Fib(n) = Fib(n-1) + Fib(n-2)$

↑

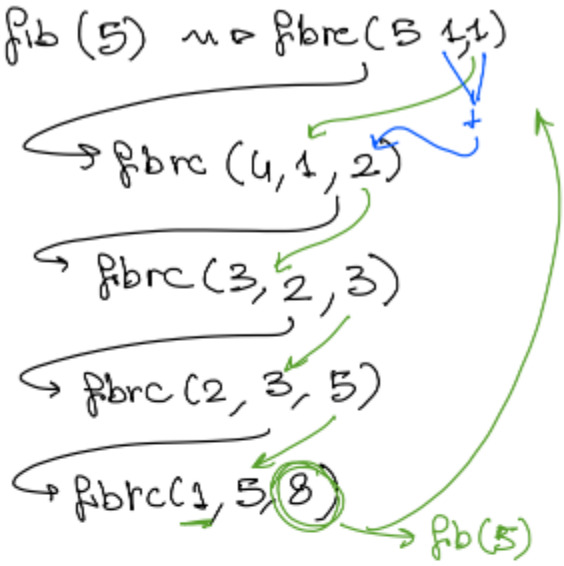
```
int fib (int n) {
    if n <= 1 return 1 ;
    else return fib(n-1) + fib(n-2) ;   NON È IN CODA
    }
```

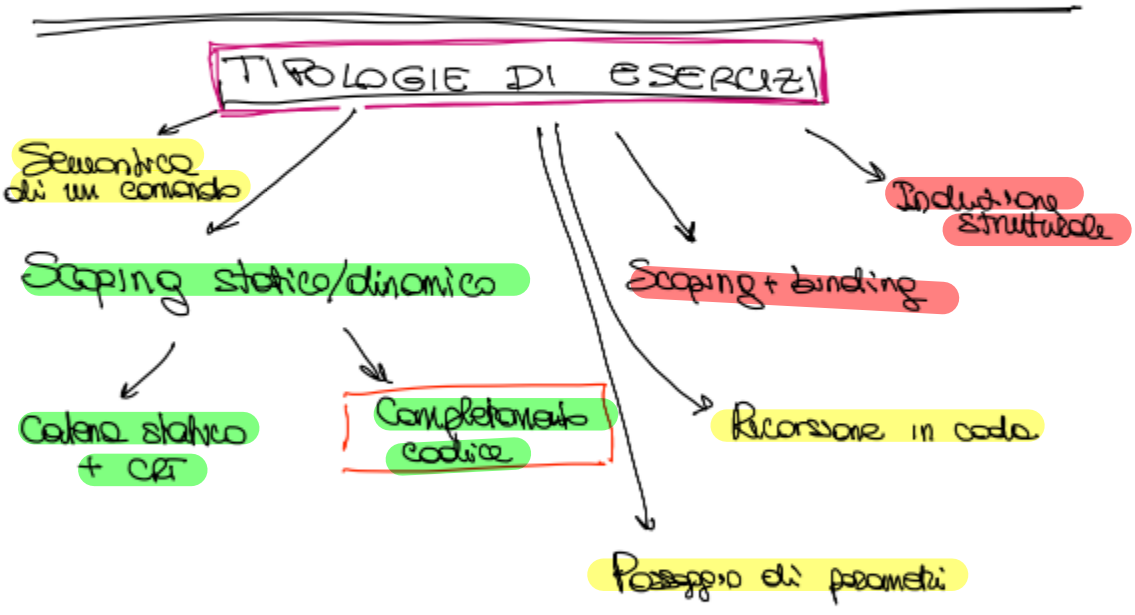
⇝ in coda

fib(n-2) fib(n-1)

```
int fib(n) { return fibrc(n, 1, 1); }

int fibrc (int n, int res1 , int res2) {
    if (n<=1 ) return res2 ;
    else return fibrc (n-1, res2, res1 + res2);
    }
```





Esercizio di scoping di completamento del codice

```
{ int i;
  for (int i=0; i<=1; i++){
      int x;
      x= fun ();
  }
```

Fornire il codice da inserire in (*) in modo tale che

1. Se il linguaggio ha scoping statico le due chiamate a fun restituiscono lo stesso valore
2. Se il ling. ha scoping dinamico le due chiamate a fun restituiscono valori diversi.

→ il codice deve essere eseguibile → dobbiamo definire fun()

→ Perché il comportam. di un programma sia potenzialmente diverso in dipendenza delle regole di scoping

è NECESSARIO che ci sia

→ una procedura con ambiente non locale

→ il riferimento non locale deve esistere sia nel contesto di definizione che in quello di chiamata

→ i è un buon candidato per essere ambiente non locale per fun

se non è inizializzato nell'ambiente di chiamata → non si può usare

(*) `i = 1;` ← serve per rendere il codice eseguibile
`int fun() { return i; }`

Esecuzione con scoping statico

↳ la chiamata di fun() restituisce la i

non viene modificata
⟹ restituisce 1 ad entrambe le chiamate

Esecuzione con scoping dinamico

↳ le due chiamate di fun() fanno il ciclo for che cambia tra una chiamata e l'altra

prima chiamata → return 0
seconda chiamata → return 1

Esercizio 2

```
{ int x;
  (X)
  void exec (){
      for (int j = 2; j <= 3; j++){
          int y = 1;
          x = fun ();
      }
  }
  exec();
}
```

codice t.c. fun() restituisca lo stesso valore nelle due chiamate con scoping statico; restituisca valori diversi con scoping dinamico

→ Dobbiamo definire fun() con ambiente non locale che possa dipendere dalle regole di scoping.

↳ uniche variabili candidate sono y e j perché l'ambiente di chiamata non è modificabile tra le due j cambia ad ogni esecuzione quindi permetterebbe di rispettare la richiesta.

→ Dobbiamo definire anche una j globale che non viene modificata

⇒ (*)
```
int j = 1
int fun () { return j; }
```

Scoping statico

La chiamata ad ogni iterazione restituisce la j globale che vale 1

Scoping dinamico

La chiamata restituisce la j del ciclo for che vale 2 alla prima iterazione e 3 alla seconda.

Esercizio 3

Ipotizziamo come nel primo esercizio che il blocco del while possa definire un ambiente locale separato da quello globale

```
int a=0;
(*)
while (a<=1) {
    int x;
    (**)
    x= fun ();
    a++;
}
```

Richiesta come per esercizio precedente

Definiamo fun() per renderlo eseguibile e fun() deve contenere un ambiente non locale per dipendere dallo scoping.

(*)
```
int x = 2;
int fun () { return x; }
```

Scoping statico → L'ambiente non locale per fun è quello globale quindi le due chiamate restituiscono 2 (valore di x non modificato)

Scoping dinamico → x non è

la prima chiamata restituisce $x = a = 0$
la seconda chiamata restituisce $x = a = 1$

Fare:

```
{ int a=0; int y=1;

  (*)

  void exec(){ int x;
               (**)
               y = fun();
               a++;
             }
  while (a<=1) {exec(); }
}
```

richieste esercizi precedenti