

Ricorsione di Coda (Tail Recursion)

Inefficienza Ricorsione

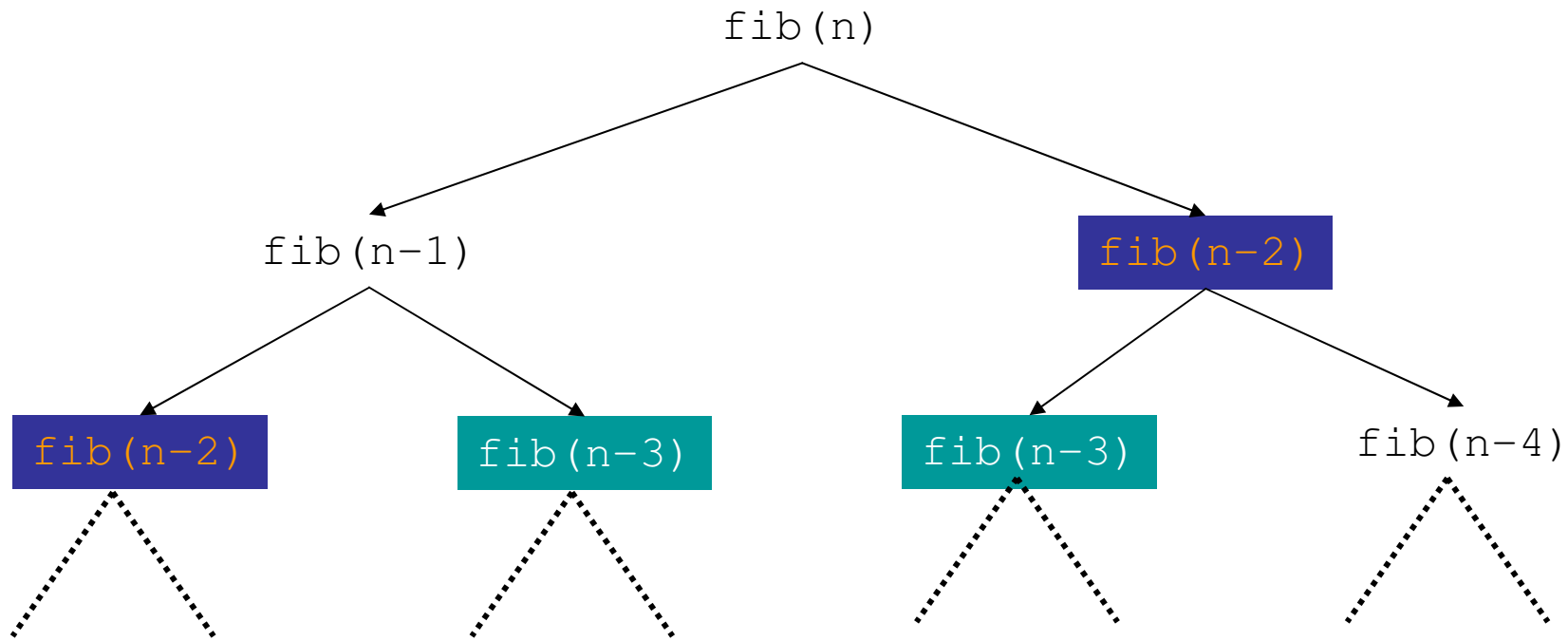
Perchè la ricorsione può risultare un implementazione inefficiente ?

La ragione risiede nel fatto che durante l'esecuzione viene creato uno STACK di RECORD di ATTIVAZIONE, per ognuna delle chiamate effettuate. Ciò potrebbe produrre un OVERFLOW di MEMORIA in tempi rapidi e consentire l'esecuzione del programma per valori di input molto piccoli.

Inefficienza Ricorsione: Fibonacci

```
int fib (int n){  
    /* Pre n >=0*/  
    if (n<=1) return 1  
    else return fib(n-1)+fib(n-2)  
}
```

Fibonacci: Albero chiamate



Si può dimostrare (Algoritmi I) che nell'esecuzione di $\text{fib}(n)$ le chiamate a $\text{fib}(0)$ e $\text{fib}(1)$ sono calcolati $\text{fib}(n+1)$ volte.

Ad esempio nella computazione di $\text{fib}(29)$ sono chiamati $832040 = \text{fib}(30)$ volte

Come ottenere efficienza di memoria

Iterazione:

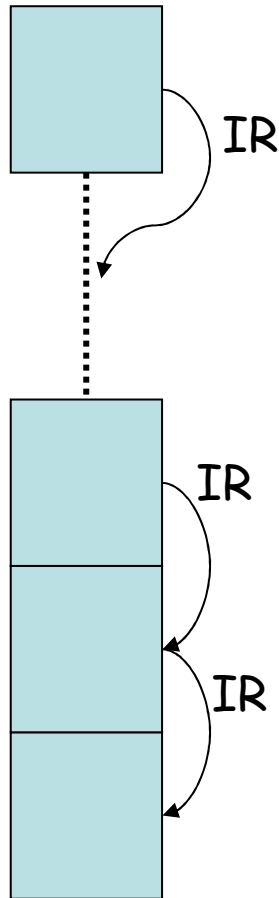
Implementando un algoritmo mediante l'iterazione in fase di esecuzione non vi sarà alcuno spreco di memoria dovuto alla crescita della STACK dei RECORD DI ATTIVAZIONE.

Ricorsione di Coda:

Idea: **Costruire una ricorsione definita in modo tale che nel caso ricorsivo l'ultima operazione da eseguire è la chiamata ricorsiva. Perché questo aiuta ?**

Idea Ricorsione di Coda: Stack

Se la chiamata ricorsiva è l'ultima operazione effettuata nella funzione chiamante, allora non ho più bisogno di specificare un indirizzo di ritorno in nella funzione chiamante e posso sostituirlo con l'indirizzo di ritorno della funzione principale da cui la ricorsiva veniva chiamata.



Esempio

```
int somma (int n){
    /* Pre n >=0*/
    if (n=0) return 0;
    else return n+somma(n-1);
}
```

```
int somma (int n, int m){
    /* Pre n >=0*/
    if (n==0) return m;
    else return somma(n-1,n+m);
}
```

Definizione

In un funzione F , una chiamata ad una funzione G si dice

Chiamata di Coda o Terminale

se in F dopo la chiamata non vi è alcuna altra istruzione da eseguire.

Si faccia attenzione che questo non vuol sempre significa che la chiamata è l'ultima cosa scritta nella funzione

Esempio

```
int F (int x1, ..., float xn) {  
    istruz 1;  
    istruz 2;  
    .....  
    .....  
    return G(y1, ..., yk);  /* CHIAMATA TERMINALE */  
}
```

```
int F (int x1, ..., float xn) {  
    istruz 1;  
    istruz 2;  
    .....  
    s= G(y1, ..., yk);  /* CHIAMATA NON TERMINALE */  
    istruz k;  
    return s;  
}
```

Esempio

```
int F (int x1, ..., float xn) {
    istruz1;
    s= G(y1, ...,yk); /* CHIAMATA NON TERMINALE */
    istruz2;
    .....
    if (.....) {
        istruz 1.1
        .....
        return G(x-y, ...); /* CHIAMATA TERMINALE */
    } else {
        s=G(x+y, ...,); /* CHIAMATA NON TERMINALE */
        istruz2.1
        .....
    }
}
```

Esempio

Ricordare che in C le istruzioni:

```
s = G(x+y,...)
```

```
return s;
```

possono semplificarsi in

```
return G(x+y,....)
```

quindi di fatto anche nel primo caso possiamo considerare la chiamata a G una chiamata Terminale

Esempio

```
int dec(int n){  
    if (n %2==0) return decpari(n)  
    else return decdisp(n);  
}
```

decpari e decdisp sono chiamate terminali