ALGORITMI RICORSIVI

Un algoritmo si dice ricorsivo quando è def. in termini di se stesso. L'algo richiama se stesso generando una sequenza di chiamate che ha termine al verificarsi di una condizione di terminazione in base a particolari valori di input.

È costituito da 2 parti:

(3)
1) Passo base: stabilisce il risultato per valori precisi dei dati iniziali
   = CONDIZIONE DI TERMINAZIONE
2) Passo di induzione: si calcola il risultato per la dimensione n dei dati in funzione del risultato per la dim. $n-1$ dei dati

(5)
ESEMPI DI ALGO RICORSIVI

Potenza: def iterativa → $a^b = \underbrace{a \cdot a \cdots a}_{b \text{ volte}}$

def ricorsiva → $a^b = \begin{cases} 1 & \text{se } b=0 \\ a \cdot a^{b-1} & \text{se } b \neq 0 \; (>) \end{cases}$

Fattoriale: def iterativa → $n! = n \cdot (n-1) \cdots 1$

def ricorsiva → $n! = \begin{cases} 1 & \text{se } n=0 \\ n(n-1)! & \text{se } n \neq 0 \; (>) \end{cases}$

---

(1) PROCEDURA RICORSIVA: quando
- all'interno della propria def. compare almeno una chiamata alla procedura stessa, oppure
- compare la chiamata ad almeno una procedura che chiama la procedura stessa diret. o indirettamente.

(2) ALGORITMO RICORSIVO: quando si basa su procedure ricorsive.

(4) RICORSIONE E ITERAZIONE: ogni programma ricorsivo può anche essere implementato in modo iterativo.
La sol migliore, come efficienza e chiarezza del codice, dipende dal problema.

La Tecnica ricorsiva permette di scrivere algo eleganti e sintetici per molti problemi comuni anche se non sempre le sol ricorsive sono le più efficienti. Questo è dovuto al fatto che la ricorsione viene implementata utilizzando chiamate a funzioni che hanno un costo rilevante. Ciò rende in genere gli algoritmi iterativi più efficienti.

RICORSIONE:

Vantaggio: permette di scrivere poche linee di codice per risolvere un problema anche molto complesso

Svantaggio: le prestazioni.
Infatti la ricorsione genera una quantità enorme di overhead, occupando lo stack per un n° di istanze pari alle chiamate delle funzione che è necessario effettuare per risolvere il problema.
Funzioni che occupano una gran quantità di spazio in memoria, pur potendo essere implementate ricorsivamente, potrebbero dare problemi di spazio di memoria esaurito e di elevato tempo di esecuzione poiché la ricorsione impegna il processore per popolare e distruggere gli stack.
Pertanto se le prestazioni sono l'obiettivo principale del programma e/o non si dispone di sufficiente memoria, la ricorsione potrebbe non essere la scelta migliore.

1) FUNZIONI MATEMATICHE RICORSIVE: la def è espressa in termini di se stessa

Es.

- POTENZA: def iterativa $a^b = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ volte}}$

  def ricorsiva $a^b = \begin{cases} 1 & \text{se } b=0 \\ a \cdot a^{b-1} & \text{se } b \neq 0 \end{cases}$

- FATTORIALE def iterativa $n! = n \cdot (n-1) \cdot \ldots \cdot 1$

  def ricorsiva $n! = \begin{cases} 1 & \text{se } n=0 \\ n(n-1)! & \text{se } n>0 \end{cases}$

- Caso base per far terminare la ricorsione altrim. si applicherebbe la def ricorsiva all'infinito
- Calcolo il valore della funzione per l'input n, conoscendo il valore della f. per l'input n-1.

2) ALGORITMI RICORSIVI

RICORSIONE DIRETTA vs RICORSIONE INDIRETTA

A(...) → A(...)

A(...) → B(...) ; B(...) → A(...)

oppure

B(...) → C(...) ; C(...) → A(...)

- TECNICA DIVIDE ET IMPERA

ESEMPI: FATTORIALE(n)

```
if n == 0
    return 1
else if n > 0
    return n * FATTORIALE(n-1)
else return UNDEFINED (NULL)
```

se non gestisco questo caso con input negativo la ricorsione non termina!

• Applicazione algoritmo per calcolare il FATTORIALE (4)

FATT(4)
↳ 4 * FATT(3) = 4*6 = 24
↳ 3 * FATT(2) = 3 * 2 = 6
↳ 2 * FATT(1) = 2 * 1 = 2
↳ 1 * FATT(0) = 1*1

Termina ricorsione ← CASO BASE

4 × 3 × 2 × 1
12 × 2
24 × 1
24

Ogni algo ricorsivo può sempre essere riscritto in forma iterativa. Il viceversa non è sempre possibile e neppure consigliabile xchè gli algoritmi ricorsivi sono sempre meno efficienti in termini di tempo e spazio

CHIAMATA DI FUNZIONE:

1) Il programma scrive i parametri della funzione sullo stack in memoria e chiama la funzione
2) Il codice della funzione è caricato in memoria per l'esec
3) Sullo stack vengono memorizzati i valori delle variabili locali.

Finchè la funzione non termina, questo spazio in memoria rimarrà occupato e siccome, come visto, in un algo ricorsivo vengono effettuate chiamate ricorsive della f. per ogni chiamata si avrà ulteriore spazio occupato che sarà liberato solo quando si inizierà a risalire la catena di chiamate dopo aver raggiunto il caso base. ⇒ occupazione in spazio elevata
Il tempo in più dipende dal tempo necessario per caricare i dati sullo stack per ogni chiamata e poi ripulire lo stack al termine.

Versione iterativa fattoriale:

FATTORIALE(n)

if n<0
 return UNDEFINED (NULL)
else {
 fatt = n (≥)
 for i = n-1 to 1
  fatt = fatt * i
} return fatt

FATTORIALE (4)
fatt = 4
i = 3 fatt = 4 * 3 = 12
i = 2 fatt = 12 * 2 = 24
i = 1 fatt = 24 * 1 = 24

Se voglio proprio simulare il calcolo fatto dalla versione ricorsiva

fatt = 1 (≤)
for i = 2 to n
 fatt = fatt * i
return fatt

FATTORIALE (4)
fatt = 1
i = 2 fatt = 1 * 2 = 2
i = 3 fatt = 2 * 3 = 6
i = 4 fatt = 6 * 4 = 24

Complessità comput. asintotica:

$$\sum_{i=2}^{n} O(1) = (n-2+1)\ O(1) = \underbrace{(n-1)}_{O(n)}\ O(1) = O(n)\ O(1) = O(n)$$

(3) Come si calcola la complessità computazionale asintotica degli algoritmi ricorsivi?

Primo passo: scrivere l'equazione di ricorrenza (relazione) che descrive il tempo di calcolo $T(n)$:

- In un algoritmo ricorsivo bisogna distinguere il costo del caso base dal costo derivante dalle chiamate ricorsive.

$$T(n) = \begin{cases} O(1) & \text{se } n = 0 \\ T(n-1) + O(1) & \text{se } n > 0 \end{cases} \rightsquigarrow O(n)$$

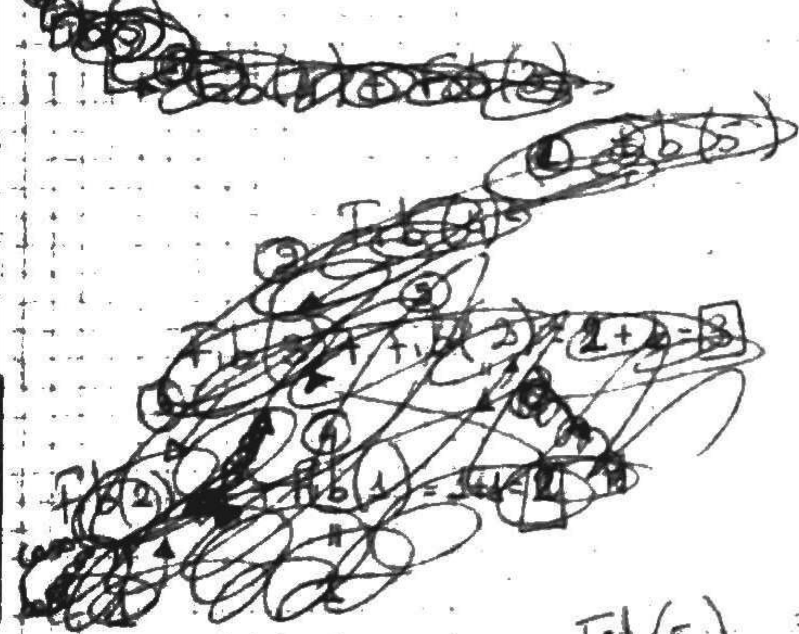
Bisogna distinguere i costi delle istruzioni che non dipendono dalle chiamate ricorsive e il 1° e il costo delle chiamate ricorsive

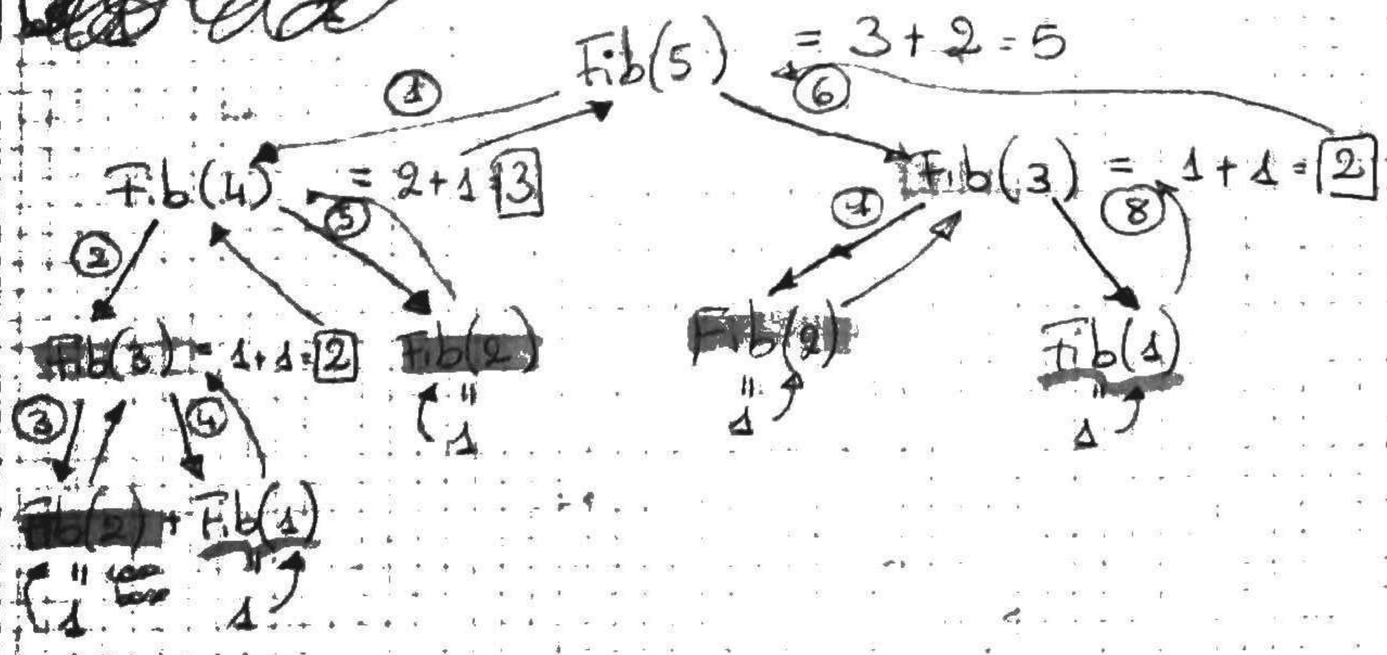
ESEMPIO: Calcolo dei numeri di Fibonacci

$$Fib(n) = \begin{cases} 1 & se\ n = 1, 2 \\ Fib(n-1) + Fib(n-2) & se\ n \geq 3 \end{cases}$$

```
FiB(n)
  if (n == 1 OR n == 2)
      return 1
  else if n >= 3
      return Fib(n-1) + Fib(n-2)
  else return UNDEF (null)
```

• Applicazione per calcolare FiB(5)





Uno stesso calcolo viene ripetuto molte volte!
Questo xche per risolvere un probl su dim n dobbiamo risolvere altri due probl di dim poco inferiore: $n-1$ e $n-2$.

Equazione di ricorrenza:

$$T(n) = \begin{cases} O(1) & \text{se } n=1 \text{ o } n=2 \\ T(n-1)+T(n-2)+O(1) & \text{se } n \geq 3 \end{cases}$$

$$\Downarrow$$

$$O(2^n)$$

Questo è un caso in cui la versione ricorsiva ha complessità asintotica ~~maggiore~~ superiore rispetto alla versione iterativa.

```
FiB(n)
  if(n== 1 || n== 2)
    return 1
  else if n ≥ 3
  {
    prec_prec = 1
    prec = 1
    for i = 3 to n-1
    {
      fib = prec + prec_prec
      prec_prec = prec
      prec = fib
    }
    fib = prec + prec_prec
    return fib
  }
  else return UNDEF
```

FiB(5)

i = 3 fb = 1+1 = 2
pp = 1
p = 2
i = 4 fb = 2+1 = 3
pp = 2 p = 3
i = 5 fb = 3+2 = 5

Compl asintotica $T(n) = \sum_{i=3}^{n-1} O(1) + O(1) =$

$= (n-1-3+1) \cdot O(1) + O(1) = (n-3) \cdot O(1) + O(1)$ (dove $(n-3) = O(n)$)

$= O(n) \cdot O(1) + O(1) = O(n) + O(1) = O(n)$

ESERCIZIO: SOMMA DEGLI EL DI UN ARRAY

RICERCA DICOTOMICA: ALGORITMO RICORSIVO

```
RICERCA_BIN (A, k, inizio, fine)
  if inizio > fine
     return -1
  else {
       medio = ⌊(inizio + fine)/2⌋
       if A[medio] == k
          return medio
       else if A[medio] > k
          return RICERCA_BIN(A, k, inizio, medio-1)
       else return RICERCA_BIN(A, k, medio+1, fine)
  }
```

$$T(n) = \begin{cases} O(1) & \text{se } n = 0 \; [\text{o } n = 1] \\ T\left(\frac{n}{2}\right) + O(1) & \text{se } n > 1 \end{cases} \Rightarrow O(\log_2 n)$$

$A = [\overset{0}{1}, \overset{1}{2}, \overset{2}{3}, \overset{3}{4}, \overset{4}{5}]$ $k = 4$

RIC_BIN(A, 4, 0, 4) ← RETURN 3

① └→ RIC_BIN(A, 4, 3, 4) ← RETURN 3

└→ RETURN 3