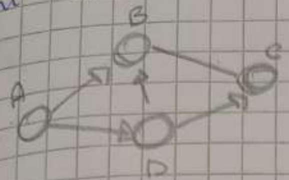
OTTIMIZZAZIONE

Un problema è di ottimizzazione quando esistono varie soluzioni = ma solo alcune di queste sono le migliori



Due soluzioni:

A - B - C — soluzione migliore

A - D - B - C

Per capire quale soluzione è la migliore definiamo la funzione bontà che assegna ad ogni soluzione un grado di bontà, viene definita così $f: S \to R$

↳ insieme delle soluzioni

PROG. DINAMICA

TOP-DOWN → BOTTOM-UP

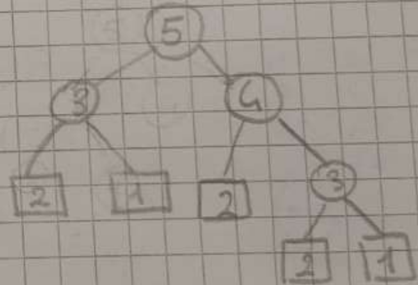
ricorsivo → iterativo

Esempio:

Fibonacci → $F_n = \begin{cases} 1 & \text{se } n \leq 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$

modo ricorsivo

albero di ricorsione con n 5




```
F(n):
  if n ≤ 2 then
      return 1
  else
      return F(n-2) + F(n-1)
```

$O(2n)$

Questo algoritmo ha dei problemi: infatti i sottoproblemi vengono risolti inutilmente più volte come il 3 nel nostro albero. Come lo evitiamo?

Ogni volta che un sottoproblema viene risolto diventa un caso base.

nuova funzione →

```
M = new Array(n)   } array che all'iesimo posto
                   } contiene la soluzione al problema i
M[1] = M[2] = 1      inizializzazione
for i = 3 to n DO M[i] ← NULL

F(n)
  if M[n] ≠ null then
     return M[n]
  if M[n-1] = null then
        M[n-1] ← F(n-1)
  if M[n-2] = null
        M[n-2] ← F(n-1)
  return M[n-1] + M[n-2]
```

hanno le stesse istruzioni

Entrambe le soluzioni sono top-down (dai problemi più piccoli a quelli più grandi

Bottom-up di seguito:

```
F(n):
  M[1] ← M[2] ← 1
  for i ← 3 to n do
     M[i] ← M[i-2] + M[i-1]
  return M[n]
```

L'approccio ricorsivo puro non ha senso se un sottoproblema deve essere risolto più volte per arrivare alla soluzione, in quel caso si deve usare la <u>memorizzazione</u>. Se lo spazio delle soluzioni viene esplorato tutto allora usiamo un approccio bottom-up, sennò top down.

Problema di ottimizzazione ROD - CUT

Abbiamo un azienda che taglia delle ROD con i seguenti guadagni:

i	1	2	3	4	5	6	7	8	9	10
$p(i)$	1	5	8	9	10	17	17	20	24	30

$\square\square\square\square \rightarrow 9$

$\square\,\square\square\square \rightarrow 1+8=9$

$\square\square\,\square\square \rightarrow 5+5=10$ ← soluzione migliore

$\square\,\square\square\,\square \rightarrow 1+5+1=7$

$\square\,\square\,\square\,\square \rightarrow 4$

} possibili tagli di una barra da 4

Come cercare una soluzione:

- 1 Sottostrutture
  - sottostrutture ottime?
  - approccio ricorsivo?
- 2 Definisco una funzione ricorsiva per il calcolo del costo di una sol ottima
- 3 Costruire una procedura bottom-up per il calcolo del di una sol ottima
- 4 Costruzione di un sol ottima OPZIONALE

n

$k \;/\; n-k$

$P(n) \rightarrow S(n)$

$P(n) \leftarrow P(k) + P(n-k)$ $\quad k < n$, $n-k < n$, $1 \leq k \leq n$

$S(n) \leftarrow S(k) + S(n-k)$ una soluzione ottima è fatta da sottoproblemi ottimi?

$S^*(n) = S(k) + S^*(n-k) \leq$

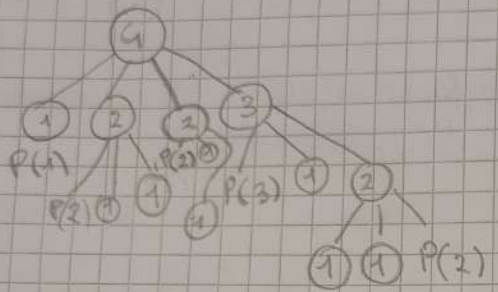
$\leq S^{**}(n) = S^*(k) + S^*(n-k) \quad \rightarrow \quad S^*(n) \leq S^{**}(n)$

$S^*(n) \geq S^{**}(n)$

definizione funzione →

$$r(i) = \begin{cases} 0 & \text{se } i = 0 \\ \max\limits_{1 \le k \le i} \underbrace{(r(k) + r(i-k))}_{\to\ 2n \text{ chiamate ricorsive}} & \text{se } i \ge 1 \end{cases}$$

lunghezza ↑ i



Serve la memorizzazione perché alcuni sottoproblemi si ripetono

	1	2	3						n-1	n
R										

```
ROD-CUT(n)
  if n=0 then
     return 0
  for i ← 1 to n
     m ← P(i)
     for k ← 1 to i-1 DO
        if R(k)+R(i-k) > m
        then m ← R(k)+R(i-k)
  r[i] ← m
```

stiamo dando per scontato che esistono P e R due array

↳ valori lunghezze → così già esaminati

praticamente il calcolo del massimo per le barre di lunghezza i

Esempio con P uguale a quello scritto all'inizio

	1	2	3	4
R	1			
	1	5		
	1	5	8	
	1	5	8	10

passi di iterazione con 4

$1\ 5\ 8\ 10$ → valore ottimale

$O(n^2)$

Definiamo K un array che ci dice il miglior modo per suddividere le barre di dimensione i

1	2	3	4
1	2	3	2

→ il taglio viene fatto nel punto 2

```
ROD-CUT(n)
  R ← new Array(n)
  K ← new Array(n)
  if n=0 then
     return 0
  for i ← 1 to n
     m ← P(i)
     K[i] ← i
     for k ← 1 to i-1 DO
        if R[k] + R[i-k] > m
        then m ← R[k] + R[i-k]
             K[i] ← k
     R[i] ← m
```

```
PRINT-CUT(n, K)
 if K[n] = n
    print(n)
else
   print-cut(K[n], K)
   print-cut(n - K[n], K)
```