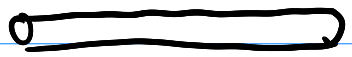
PROGRAMMAZIONE DINAMICA

- Una tecnica algoritmica che può essere applicata a molti problemi computazionali, tipicamente di OTTIMIZZAZIONE

Vediamo un primo caso di studio

TAGLIO DEL BASTONE



Abbiamo un bastone lungo 4 di materiale

Possiamo vendere segmenti di lunghezza i

i	1	2	3	4
ricavo	1	5	8	9

-  → ricavo 4
-  → ricavo 9
-  → ricavo 7
-  → ricavo 10 ← TAGLIO OTTIMO

Il ricavo non è proporzionale alla lunghezza altrimenti i tagli non farmo differenza

Q: Data una barra di lunghezza n e un array P dove $P[i]$ indica il ricavo alla vendita di una barra di lunghezza i

Trovare un TAGLIO OTTIMO che massimizzi il ricavo totale

Quindi se scrivo $n = i_1 + i_2 + i_3 + \dots + i_k \quad i_j \in \mathbb{N} \setminus \{0\}$

i ricavo corrispondente è $P[i_1] + P[i_2] + \dots + P[i_k]$

OSS I modi di tagliare una barra di lunghezza n sono

$\boxed{2^{n-1}}$ perché ho $n-1$ scelte binarie di tagliare a distanza i

Dati n e P possiamo quindi definire $R(t)$ come il ricavo massimo che si può ottenere con una barra di dimensione

$$t \leq n$$

SOTTOSTRUTTURA OTTIMA DELLE SOLUZIONI

$R(0) = 0$ $\qquad$ $R(1) = P[1]$

per $t > 1$ allora possiamo fare

0 tagli $\longrightarrow$ ricavo $P[t]$

≥ 1 taglio $\longrightarrow$ ricavo $P[k] + R(t-k)$ se il primo taglio è in posizione k

quindi

$$R(t) = \max\left(P[t], P[1] + R(t-1), P[2] + R(t-2), \dots\right)$$

riscritto

$$= \max_{1 \leq k \leq t} \left(P[1] + R(t-k)\right)$$

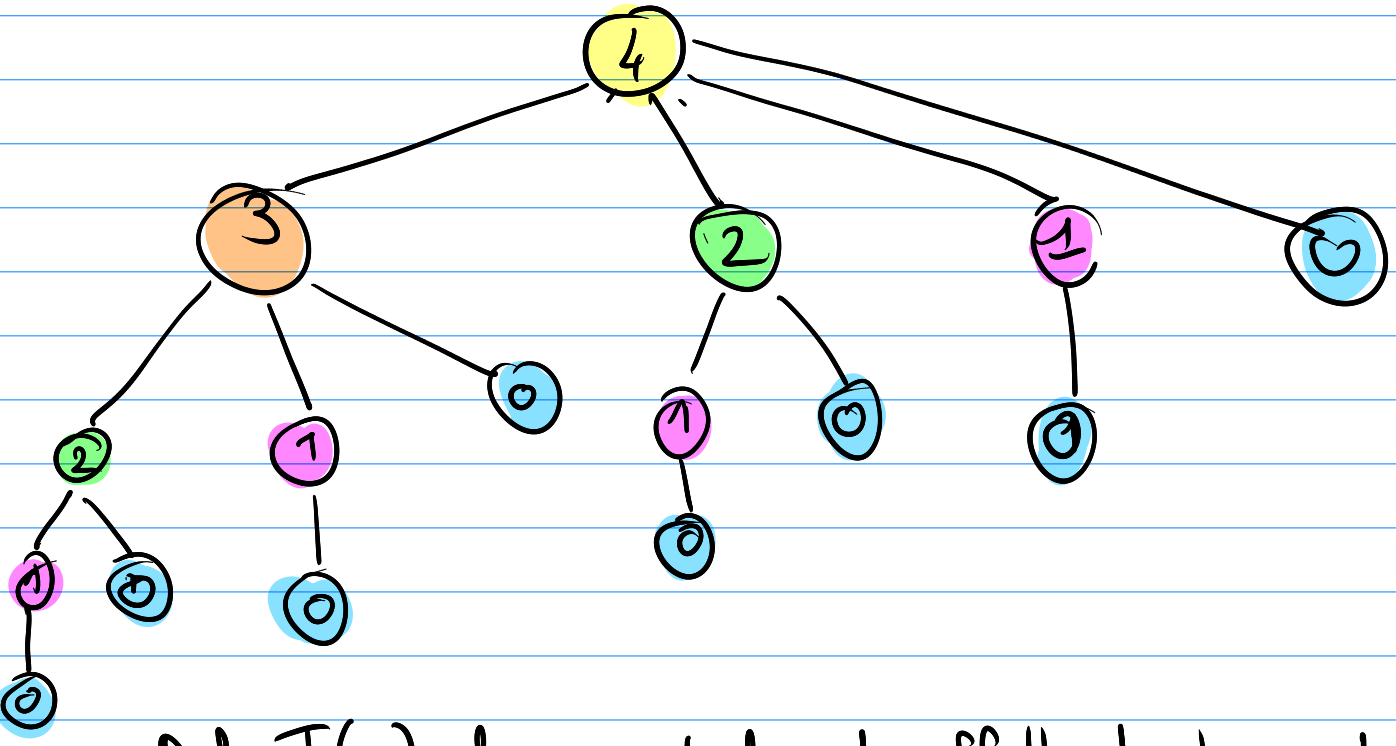
caso $k = t$ è 0 tagli con $R(0) = 0$

IMPLEMENTAZIONE RICORSIVA

- Vediamo un implementazione ricorsiva top-down di un algoritmo che usa questa formula per risolvere il problema

Vediamo: funzione 'max_cut_rod' nel file 'code_rod.py'

Questa funzione ricorsiva, per $t = 4$ ha il seguente albero di chiamate



Def: $T(n)$ il numero di chiamate effettuate chiamando la funzione per $t = n$

$$T(0) = 1$$

$$T(n) = 1 + \sum_{k=1}^{n} T(n-k) = 1 + \sum_{k=0}^{n-1} T(k)$$

Esercizio (15.1-1): dimostrate che $T(n) = 2^n$

La nostra funzione ricorsiva fa 2^n chiamate. Vediamo la stessa funzione con un contatore che tiene traccia

Vediamo: funzione 'max_cut_rod_cnt' nel file 'code_rod.py'

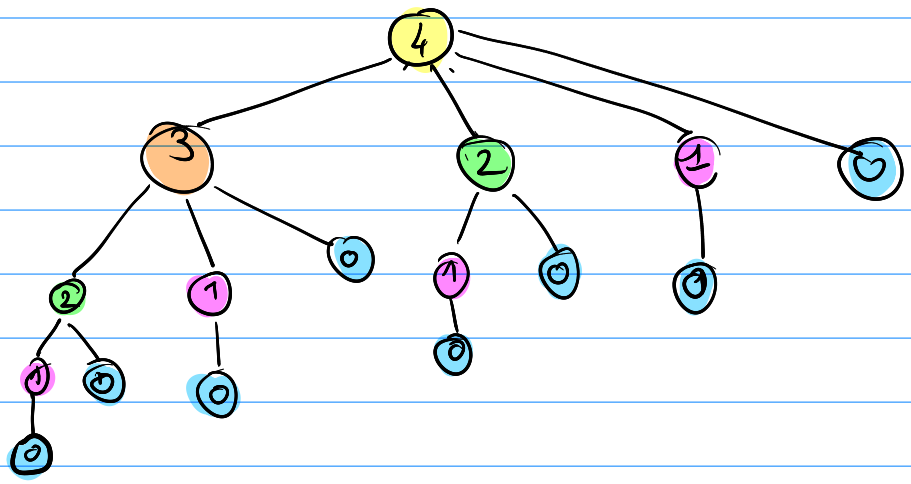
Possiamo migliorare MOLTO il tempo di esecuzione osservando che nello schema della ricorsione molte chiamate sono ripetute

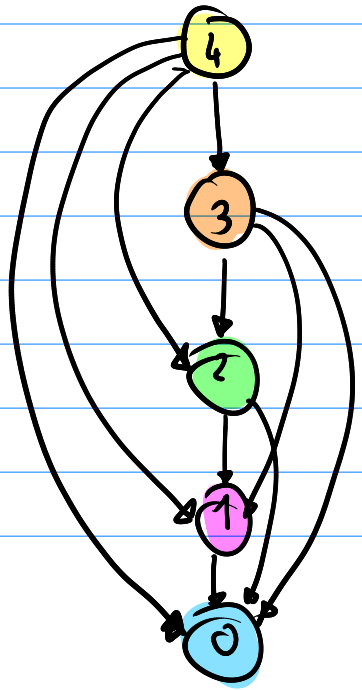
(Memoization) Annotazione delle chiamate

- La tecnica consiste nel "salvare" il risultato di chiamate già fatte e di utilizzare il valore salvato invece di ricalcolare.
- In un algoritmo ricorsivo questa ha un effetto potente perché ogni chiamata evitata evito tutte le sotto chiamate

Vediamo: funzione 'max_cut_rod_memoization' nel file 'code_rod.py'

Lo schema delle chiamate è cambiato





Per ogni t il calcolo viene fatto una volta sola e richiamato più volte

OSS le chiamate sono $\geq \frac{n^2}{2}$ per calcolare $R(n)$

- ogni $R(i)$ esegue il ciclo di chiamate una volta sola
- il ciclo di chiamate in $R(i)$ ha complessità $O(i)$

In totale, per $t=n$ #chiamate $\geq \frac{n^2}{2}$

Interpretazione bottom-up dell'algoritmo

Possiamo calcolare l'array $[R(1), \dots, R(n)]$ andando in senso inverso

Per $t = 0, \dots, n$ calcoliamo $R(t)$ usando i valori già noti di $R(i)$ i<t

EVITANDO le chiamate di funzione e la gestione della MEMOIZZAZIONE.

```
def max_cut_rod_bu(P):
    R=[0]*len(P)
    for t in range(1,len(P)):
        for k in range(1,t+1):
            R[t] = max(R[t],P[k] + R[t-k])
    return R
```

La complessità è $\Theta(n^2)$ come per la versione top-down con memoizzazione.

CALCOLO DELLA SOLUZIONE

- Abbiamo calcolato il ricavo ottimo $R(t)$.
- Ma a quale taglio corrisponde?

- In $S[t]$ memorizziamo il PRIMO TAGLIO
- I restanti sono quelli della soluzione di $R(t-k)$.

```
def max_cut_rod_bu(P):
    R=[0]*len(P)
    S=[0]*len(P)
    for t in range(1,len(P)):
        for k in range(1,t+1):
            v = P[k] + R[t-k]
            if v>R[t]:
                R[t] = v
                S[t] = k
    return R,S
```

ESERCIZIO (15.1-2)

- Consideriamo la strategia greedy che (1) sceglie di effettuare il primo taglio alla posizione k per cui il ricavo per unità $\frac{p_k}{k}$ è massimo (2) prosegue sulla barra residua

- Trovate un controesempio che mostra che questa non è una strategia ottima

ESERCIZIO (15.1-3) Assumete che ogni taglio abbia un costo c.

Descrivete un algoritmo che calcoli il ricavo massimo in questo caso.

Fasi della definizione di un algoritmo basato sulla PROGRAMMAZIONE DINAMICA

1. Caratterizzazione della struttura di una soluzione ottima
2. Soluzione ricorsiva al problema
3. Calcolo Bottom-up del valore ottimo
4. Calcolo della soluzione ottima

- Vediamo queste quattro fasi con il problema seguente

PARENTESIZZAZIONE OTTIMA

- Date tre matrici A, B, C di dimensioni

$A \to 10 \times 100$ $\quad B \to 100 \times 5$ $\quad C \to 5 \times 50$

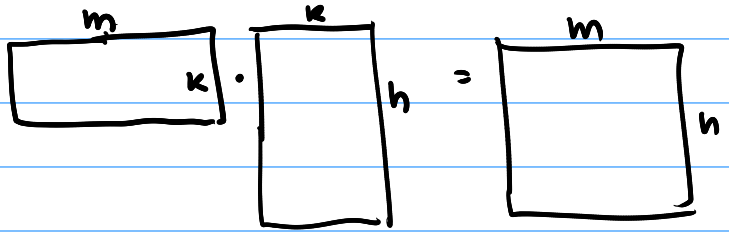
allora il prodotto $A \cdot B \cdot C$ è ben definito ed è una matrice 10×50

- Per la proprietà associativa

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

tuttavia PARENTESIZZAZIONI DIVERSE richiedono un NUMERO DI MOLTIPLICAZIONI diverse.

Supponiamo di usare l'algoritmo standard per il prodotto di matrici



in $m \cdot k \cdot h$ operazioni

allora

$(A \cdot B) \cdot C$ richiede $(A \cdot B) \cdot C$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$

$A \cdot (B \cdot C)$ richiede $A \cdot (B \cdot C)$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$

La parentesizzazione influisce sul numero di moltiplicazioni anche di 10 volte!

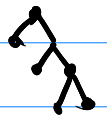
PROBLEMA: date $A_1, A_2, \dots, A_n$ matrici
con dimensioni $p_0\ p_1\ p_2 \dots p_n$

dove A_i ha dimensioni $p_{(i-1)} \times p_i$

Va trovata una parentesizzazione che minimizza il numero di moltiplicazioni necessarie a calcolare

$$A_1 \cdot A_2 \cdot A_3 \cdot \dots \cdot A_n$$

Esempio Per 4 matrici $A_1\ A_2\ A_3\ A_4$ ci sono le seguenti possibilità


$\cdot (A_1(A_2(A_3 A_4)))$ $\cdot ((A_1(A_2A_3))A_4)$


$\cdot (A_1((A_2A_3)A_4))$ $\cdot (((A_1A_2)\cdot A_3)\cdot A_4))$


$\cdot ((A_1A_2)(A_3A_4))$

In generale per moltiplicare n matrici il numero di parentesizzazioni è

$$P(n) = \begin{cases} 1 & \text{se } n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n \geq 2 \end{cases}$$

poiché ogni parentesizzazione calcola il prodotto finale tra due matrici:

OSS $P(n) = \Omega(4^n / n^{3/2})$

ESERCIZIO (15.2-3) Mostrare che $P(n) = \Omega(2^n)$

Quindi non è possibile tentare tutte le parentesizzazioni per trovare la migliore

① STRUTTURA DI UNA SOLUZIONE OTTIMA

- Notazione: $A_{i,j}$ è il risultato del prodotto $A_i \cdot A_{(i+1)} \cdot \dots \cdot A_j$
  $A_{i,i} = A_i$
- Il prodotto $A_{i,j}$ con $j > i$ è costituito da vari prodotti di cui <u>l'ultimo</u> è tra $A_{i,k}$ e $A_{k+1,j}$ per qualche $i \le k < j$.
- Dunque la parentesizzazione OTTIMA per $A_{i,j}$ contiene delle parentesizzazioni per $A_{i,k}$ e $A_{k+1,j}$ per qualche $i \le k < j$.
- Le due parentesizzazioni devono essere ottime altrimenti potremmo migliorare quella di $A_{i,j}$

② SCHEMA DI CALCOLO RICORSIVO

Un schema ricorsivo per calcolare il numero minimo di moltiplicazioni è semplice, dalle considerazioni precedenti

Se $m[i,j]$ è il minimo numero di moltiplicazioni per calcolare $A_{i,j}$, e $i < j$

$$m[i,j] = m[i,k] + m[k+1,j] + p_{(i-1)} p_k p_j$$

assumendo di sapere la divisione ottima k

Oss $A_{i,k}$ ha dim $p_{(i-1)} \times p_k$ e $A_{(k+1),j}$ ha dimensioni $p_k \times p_j$

quindi il costo dell'ultimo prodotto è $p_{(i-1)} \cdot p_k \cdot p_j$

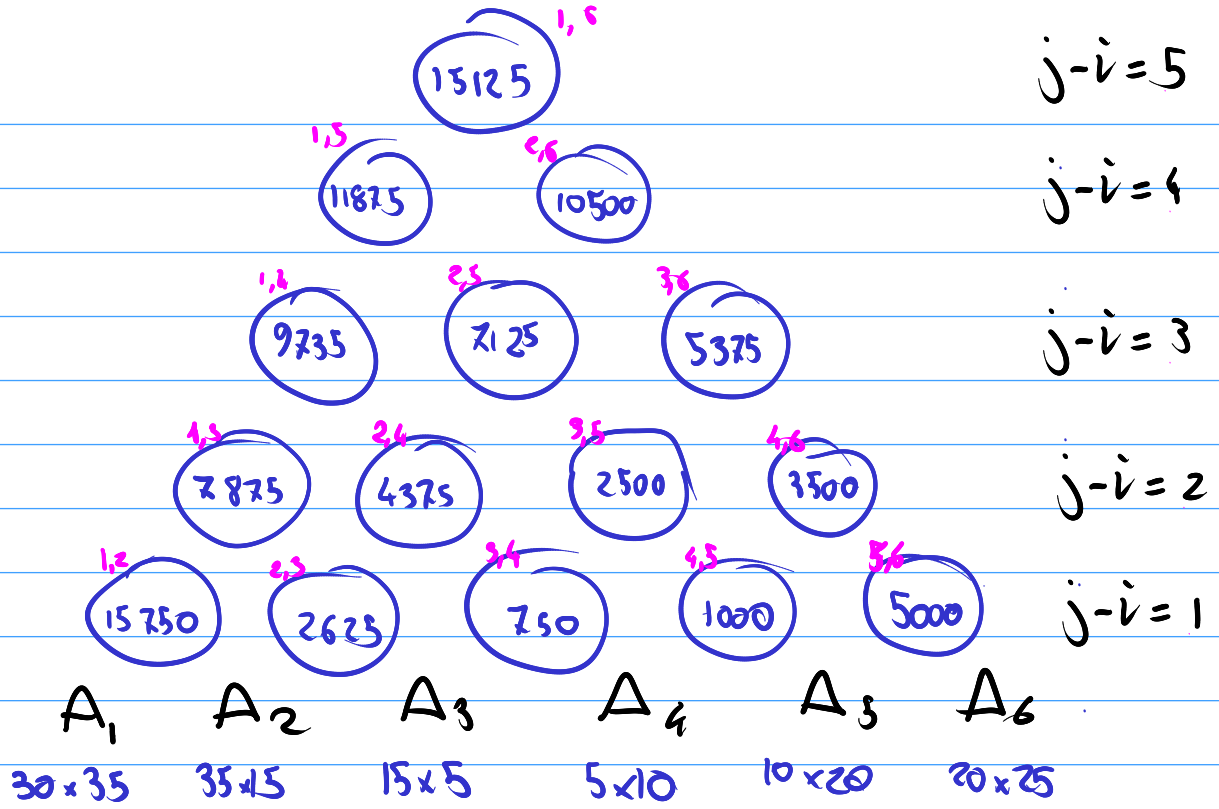
Non conoscendo k, dobbiamo trovare quello migliore

$$m[i,j] = \begin{cases} 0 & \text{per } i=j \\ \min\limits_{i \le k < j} m[i,k] + m[k+1,j] + p_{(i-1)} \cdot p_k \cdot p_j & \text{per } i<j \end{cases}$$

- Dunque abbiamo una definizione ricorsiva di $m[i,j]$
- La definizione non è circolare perché ogni prodotto $A_{i,j}$ è definito in base a prodotti di intervalli più piccoli di $(j-i)$
- Gli intervalli di lunghezza zero $A_{i,i}$ sono il caso base

③ Calcolo bottom up delle soluzioni dei sottoproblemi

Sia m un dizionario oppure una matrice $n \times n$ di cui non useremo la parte che sta sopra alla diagonale principale




```python
def bestpar(P):
    n = len(P)-1
    m={}

    for i in range(1,n+1):
        m[i,i] = 0

    for l in range(2,n+1): # lunghezza del prodotto
        for i in range(1,n-l+2):
            j = i + l - 1

            m[i,j] = inf
            for k in range(i,j):
                q = m[i,k] + m[k+1,j] + P[i-1]*P[k]*P[j]
                if q < m[i,j]:
                    m[i,j] = q
    return m
```

$(j-i) = l-1$

Uso memoria $O(n^2)$

Complessità $O(n^3)$

④ Calcolo della soluzione

- Fino ad ora abbiamo solo calcolato il numero minimo delle moltiplicazioni, ma non la parentesizzazione.
- Come già visto per avere la parentesizzazione di

$$A_i \dots A_j$$

basta memorizzare il k per cui si è deciso di definire

$$m[i,j] = m[i,k] + m[k+1,j] + p_{(i-1)} \cdot p_k \cdot p_j$$

- Il resto della parentesizzazione ottima si ottiene da quella di $A_{i,k}$ e $A_{k+1,j}$

Possiamo definire quindi $s[i,j]$ con $i \leq j$ dove

$$s[i,i] = i \quad \forall\ 1 \leq i \leq n$$

$s[i,j] = k$ per la scelta dell'algoritmo di cui sopra

```
for k = i ... (j-1)
    ...
    if q < m[i,j]
        m[i,j] = q
        s[i,j] = k
```