Programmazione Dinamica (PD) vs Divide-et-Impera(DI)

- Entrambe si basano sulla scomposizione ricorsiva di un problema in sottoproblemi
- **DI** è efficiente se i sottoproblemi sono **indipendenti** tra di loro
- Se i problemi sono dipendenti, **DI** può richiedere tempi più lunghi(spesso esponenziali)
- **PD** invece riesce a ridurli(spesso a polinomiali)

Condizioni di Applicabilità

Per applicare la **PD** occore siano verificate le seguenti proprietà:

1. **Sottostruttura della soluzione:**
   deve esserci una relazione fra le soluzioni(ottimali) dei sottoproblemi e la soluzione(ottimale) del problema
2. **Sottoproblemi ripetuti**

Fasi di Sviluppo

1. Caratterizzare la struttura della soluzione
2. Definire ricorsivamente la soluzione
3. Eliminazione delle ripetizioni mediante l'annotazione dei risultati più semplici(**memoization**), ovvero mi salvo in memoria i risultati già calcolati
4. Sviluppo di un approccio **bottom-up**(dal basso verso l'alto, dai problemi più semplici a quelli più difficili) e quindi iterativo

Successione di Fibonacci

Se provassimo a calcolare il termine n di una successione di fibonacci, potremmo usare un approccio ricorsivo:

FIB(n)
▷ Pre: $n > 0$ intero
▷ Post: ritorna l'n-mo numero della sequenza di Fibonacci
if $n \leq 2$ **then**
  $f \leftarrow 1$
else
  $f \leftarrow \text{FIB}(n-1) + \text{FIB}(n-2)$
end if
return f

Tuttavia questa soluzione ha una complessità asintotica $O(\Phi^n)$ dove $\approx 1,7$ (infatti usando il **Teorema Master** delle relazioni di ricorrenza, notiamo facilmente che è una complessità **esponenziale**)

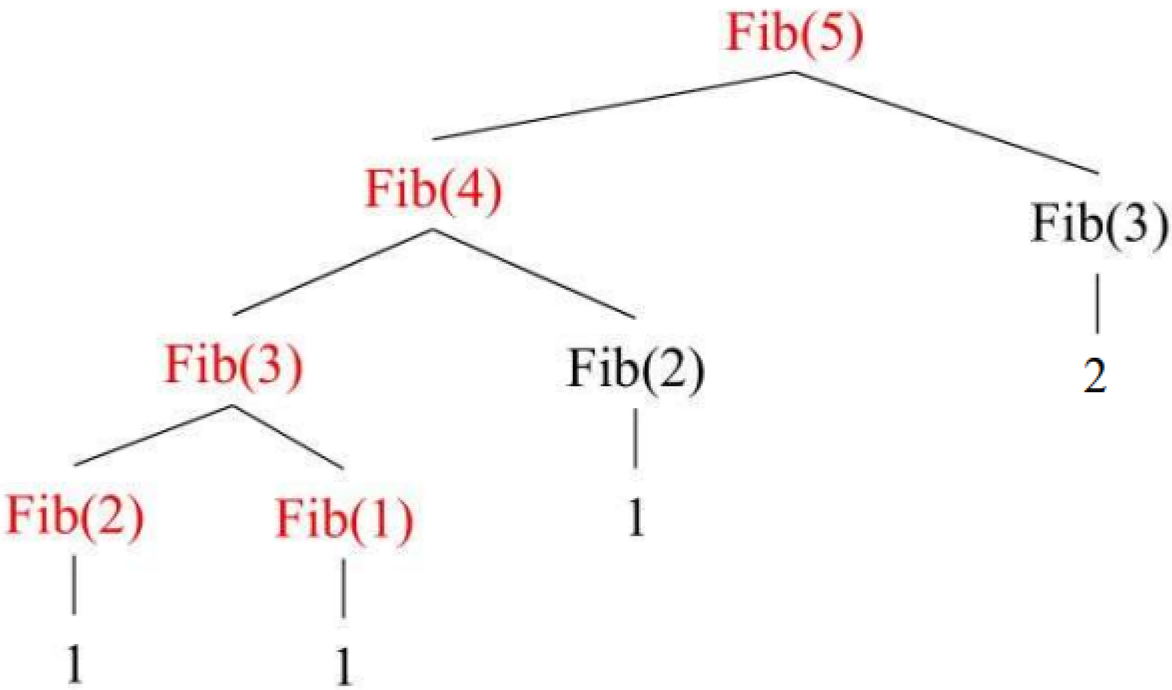
Possiamo però, salvarci i risultati di ogni chiamata ricorsiva così da non doverli ricalcolare(**memoization**)

Successione di Fibonacci mediante Memoization

FIB-MEMOIZATION(n, $memo$)
▷ Pre: $n > 0$ intero, $memo$ array di dim. $> n$
▷ Post: ritorna F_n = n-mo numero della sequenza di Fibonacci
if $memo[n] \neq nil$ **then**
  return $memo[n]$
end if ▷ $memo[n]$ non contiene alcun valore
if $n \leq 2$ **then**
  $f \leftarrow 1$
else
  $f \leftarrow \text{FIB-MEMOIZATION}(n-1, memo) + \text{FIB-MEMOIZATION}(n-2, memo)$
end if
$memo[n] \leftarrow f$
return f

Ora il metodo ha un parametro in più: **memo**.
Memo è un array(con dimensione $> n$) che contiene i valori di ogni chiamate ricorsiva.
Prima di effettuare la chiamata ricorsiva per calcolare $Fib(n)$ controllo che in $memo[n]$ ci sia già il valore. **Se** non c'è, procedo a calcolarlo e poi a salvarlo in $memo[n]$ appunto.

Lo spazio utilizzato dalla versione $Memoization$ di $Fibonacci$ per l'array è $\Theta(n)$



Albero della chiamata di Fib(5) con memoization.

Versione Bottom-up con Array

$\text{FIB-BOTTOMUP}(n)$
▷ Pre: $n > 0$ intero
▷ Post: ritorna $F_n = n$-mo numero della sequenza di Fibonacci
if $n \leq 2$ **then**
  return 1
else
  $\text{FIB}[1..n]$ sia un array di dimensione n
  $\text{FIB}[1] \leftarrow 1$, $\text{FIB}[2] \leftarrow 1$
  for $i \leftarrow 3$ **to** n **do** ▷ inv: $\forall j < i.\ \text{FIB}[j] = F_j$
    $\text{FIB}[i] \leftarrow \text{FIB}[i-1] + \text{FIB}[i-2]$
  end for
end if
return $\text{FIB}[n]$

Usando un approccio **bottom-up**, invece di partire dall'alto(e quindi

direttamente da n) partiamo dal basso calcolandoci $F(i)$ con $i \rightarrow n$.
Ogni valore di $Fib(i)$ lo salviamo in un array e lo sommiamo a $Fib(i-1)$.

Anche questa versione ha spazio e tempo computazione $\in \Theta(n)$

Versione Bottom-up senza Array

```
FIB-ITER(n)
▷ Pre: n > 0 intero
▷ Post: ritorna F_n = n-mo numero della sequenza di Fibonacci
if n ≤ 2 then
    return 1
else
    FIBA ← 1, FIBB ← 1
    for i ← 3 to n do        ▷ inv: FIBA = F_{i−1}, FIBB = F_{i−2}
        tmp ← FIBA + FIBB
        FIBB ← FIBA
        FIBA ← tmp
    end for
end if
return FIBA
```

Salvandoci in una variabile temporanea il valore precedente, possiamo anche evitare l'utilizzo di un array.
In questo caso, il tempo computazionale rimane sempre $\in \Theta(n)$ mentre invece lo spazio utilizzato $\in O(1)$

Massima Sottosequenza Comune - LCS

- date due sequenze S_1 e S_2:

$$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

$$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

- la massima sottosequenza comune è S_3

$$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

$$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

$$S_3 = \text{GTCGTCGGAAGCCGGCCGAA}$$

Sequenza di simboli che compaiono nello stesso ordine in un'altra sequenza. Nota bene: questa sequenza può essere anche interrotta

- date due sequenze X e Z:

$$X = \langle x_1, \dots x_m \rangle \qquad Z = \langle z_1, \dots z_k \rangle$$

- $Z \sqsubseteq X$, cioè Z è **sottosequenza** di X, se

$k \leq m$

$\exists f : \{1, \dots, k\} \to \{1, \dots, m\}$ crescente e t.c. $\forall j \leq k.\ z_j = x_{f(j)}$

- Z prende elementi non necessariamente consecutivi da X
- una sottosequenza Z è LCS(X, Y), cioè Z è **massima sottosequenza comune**, se

$$Z \sqsubseteq X \wedge Z \sqsubseteq Y \wedge Z \text{ ha lunghezza massima}$$

- Z in generale non è unica

Z è sottosequenza di **X** se ogni simbolo di **Z** compare nello stesso ordine in

X, anche tra interruzioni varie.
Se **Z** è una sottosequenza anche di un'altra stringa **Y**, allora si dice **Sottosequenza comune**.
Definiamo inoltre **LCS**(Longest Common Subsequence) la **sottosequenza comune** più lunga.

Introduciamo inoltre i **prefissi** ovvero i primi simboli di una stringa.
In notazione:

$$\text{sia } X = \langle x_1, \ldots, x_m \rangle \quad \text{allora:}$$

$$X_0 = \langle \rangle$$

Proprietà della sottostruttura.

Date due stringhe X e Y.

- $X_m = Y_n$
  Se entrambe finiscono per lo stesso simbolo z, allora possiamo concludere che $Z = LCS(X, Y)$ conterrà z, visto che è comune ad entrambe le stringhe.
  Dunque se $x_m = y_n$, l'ultimo elemento z_k sarà appunto $x_m = y_n$, tuttavia quale sarà il penultimo elemento, ovvero z_{k-1} ?
  Sarà $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$

- $X_m \neq Y_n$
  Se finiscono con un simbolo diverso, l'ultimo elemento di Z potrà essere x_m o y_m o qualcos'altro.
  Dunque $Z = LCS(X_{m-1}, Y)$ opppure $Z = LCS(X, Y_{n-1})$ o entrambi

Lemmi e Teorema LCS

Lemma 1. Siano $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$; se $Z = \langle z_1, \dots, z_k \rangle$ è $\text{LCS}(X, Y)$ e $x_m = y_n$ allora Z_{k-1} è $\text{LCS}(X_{m-1}, Y_{n-1})$ e $z_k = x_m$.

- **dimostrazione per assurdo di $z_k = x_m$:**
- assumiamo per assurdo

$$z_k \neq x_m$$

- allora

$$\langle z_1, \dots, z_k, x_m \rangle \sqsubseteq X \wedge \langle z_1, \dots, z_k, x_m \rangle \sqsubseteq Y \wedge |\langle z_1, \dots, z_k, x_m \rangle| > k$$

  e questo contradice al "$Z = \langle z_1, \dots, z_k \rangle$ è $\text{LCS}(X, Y)$"

Lemma 2. Siano $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$; se $Z = \langle z_1, \dots, z_k \rangle$ è $\text{LCS}(X, Y)$ e $x_m \neq y_n$ allora Z è $\text{LCS}(X_{m-1}, Y)$ oppure Z è $\text{LCS}(X, Y_{n-1})$.

- è evidente che sia cosi perché se $x_m \neq y_n$ allora l'ultimo elemento di Z o non è x_m o non è y_n e quindi l'ultimo elemento di X o l'ultimo elemento di Y non serve per trovare il loro LCS

dunque segue che

Teorema. Indicando con LCS(X,Y) una LCS di X ed Y (che in generale non è unica) e supponendo che $X = X_m$ ed $Y = Y_n$ si ha che, per $i \leq m$ e $j \leq n$:

$$\text{LCS}(X_i, Y_j) = \begin{cases} \langle\rangle & \text{se } i = 0 \text{ oppure } j = 0 \\ \text{LCS}(X_{i-1}, Y_{j-1}) \frown x_i & \text{se } x_i = y_j \\ \text{longest}(\text{LCS}(X_{i-1}, Y_j), \text{LCS}(X_i, Y_{j-1})) & \text{se } x_i \neq y_j \end{cases}$$

il quale suggerisce una **definizione ricorsiva** al problema **LCS**.
Inoltre se poniamo $k = |X| + |Y| = m + n$, la relazione di ricorrenza $T(K) = 2T(K-1) + 1$ fornisce un limite superiore al tempo di calcolo equivalente a $T(K) \in \Theta(2^k) = \Theta(2^{m+n})$