FILE 7: PROGRAMMAZIONE DINAMICA

RICETTA UNIVERSALE (4 STEP)

Step 1: Caratterizzazione Ricorsiva

Definisci la struttura di una soluzione ottima S *in funzione di soluzioni ottime S_1*, S_2*, ..., S_k* di sottoproblemi più piccoli.

Step 2: Relazione di Ricorrenza

Determina una ricorrenza del tipo:

```
c(S*) = f(c(S₁*), c(S₂*), ..., c(Sₖ*))
```

Step 3: Calcolo Bottom-Up o Memoization

- **Bottom-up**: iterativo, riempi tabella dai sottoproblemi piccoli
- **Top-down**: ricorsivo + memoization per evitare ricalcoli

Step 4: Ricostruzione Soluzione (opzionale)

Mantieni informazioni per ricostruire la soluzione ottima, non solo il costo.

CARATTERISTICHE PROBLEMI PD

1. Sottostruttura Ottima

Soluzione ottima contiene soluzioni ottime di sottoproblemi.

Esempio: LCS(X, Y) contiene LCS(X', Y') per prefissi più corti.

2. Sottoproblemi Sovrapposti

Stesso sottoproblema calcolato più volte.

Esempio: Fibonacci(n) ricalcola Fibonacci(n-2) due volte.

3. Spazio Sottoproblemi Gestibile

Numero di sottoproblemi distinti deve essere polinomiale.

Tipico: $O(n)$, $O(n^2)$, $O(n^3)$

PROBLEMA 1: LONGEST COMMON SUBSEQUENCE (LCS)

Problema

Date due stringhe X[1..m] e Y[1..n], trova la sottosequenza comune più lunga.

Nota: sottosequenza NON richiede caratteri consecutivi.

Caratterizzazione Ricorsiva

```
Sia Z = LCS(X, Y)

Caso 1: Se X[m] = Y[n]
  → Z[k] = X[m] = Y[n]
  → Z[1..k-1] = LCS(X[1..m-1], Y[1..n-1])

Caso 2: Se X[m] ≠ Y[n]
  → Z non termina con X[m] OPPURE non termina con Y[n]
  → Z = max{ LCS(X[1..m-1], Y), LCS(X, Y[1..n-1]) }
```

Relazione di Ricorrenza

```
L[i,j] = lunghezza LCS di X[1..i] e Y[1..j]

L[i,j] = {
  0                           se i=0 o j=0
  L[i-1,j-1] + 1             se X[i] = Y[j]
  max(L[i-1,j], L[i,j-1])    se X[i] ≠ Y[j]
}
```

Algoritmo Bottom-Up

```
LCS_LENGTH(X, Y, m, n)
for i = 0 to m: L[i,0] = 0
for j = 0 to n: L[0,j] = 0
for i = 1 to m:
  for j = 1 to n:
    if X[i] = Y[j]:
      L[i,j] = L[i-1,j-1] + 1
      B[i,j] = "↖"  // Diagonale
    else if L[i-1,j] ≥ L[i,j-1]:
      L[i,j] = L[i-1,j]
      B[i,j] = "↑"   // Alto
    else:
      L[i,j] = L[i,j-1]
      B[i,j] = "←"   // Sinistra
return L, B
```

Complessità: Θ(m·n) tempo e spazio

Esempio Dettagliato: X = "armo", Y = "toro"

```
Tabella L[i,j]:

      ""  t   o   r   o
  ""  0   0   0   0   0
  a   0   0   0   0   0
  r   0   0   0   1   1
  m   0   0   0   1   1
  o   0   0   1   1   2

Tabella B[i,j] (direzioni):

      ""  t   o   r   o
  ""  -   -   -   -   -
  a   -   ←   ←   ←   ←
  r   -   ←   ←   ↖   ←
  m   -   ←   ←   ↑   ←
  o   -   ←   ↖   ←   ↖

Ricostruzione da B[4,4]:
- B[4,4] = "↖" → o è comune → stampa 'o', vai a [3,3]
- B[3,3] = "↑" → vai a [2,3]
- B[2,3] = "↖" → r è comune → stampa 'r', vai a [1,2]
- B[1,2] = "←" → vai a [1,1]
- B[1,1] = "←" → fine

LCS = "ro" (lunghezza 2)
```

Versione Memoizzata (Top-Down)

```
INIT_LCS_MEMO(m, n)
for i = 0 to m:
  for j = 0 to n:
    L[i,j] = -1  // Valore sentinella
return REC_LCS(m, n)

REC_LCS(i, j)
if L[i,j] ≠ -1:  // Già calcolato
  return L[i,j]
if i = 0 or j = 0:
  L[i,j] = 0
else if X[i] = Y[j]:
  L[i,j] = 1 + REC_LCS(i-1, j-1)
else:
```

```
  L[i,j] = max(REC_LCS(i-1, j), REC_LCS(i, j-1))
return L[i,j]
```

Vantaggi memoization:

- Se X[i] = Y[j], non calcola i sottoproblemi (i-1,j) e (i,j-1)
- Può essere più veloce in pratica
- Complessità peggiore uguale: $\Theta(m \cdot n)$

PROBLEMA 2: LONGEST INCREASING SUBSEQUENCE (LIS)

Problema

Dato array X[1..n], trova la sottosequenza crescente più lunga.

Perché Strengthening?

Problema originale: LIS(X) non ha sottostruttura ottima diretta.

Problema rafforzato: LIS(X_i) = LIS che **termina in X[i]**
→ Questo HA sottostruttura ottima!

Caratterizzazione Ricorsiva

```
LIS[i] = lunghezza max sottosequenza crescente che termina in X[i]

LIS[i] = {
  1                                         se i = 1
  1 + max{LIS[j] : j < i AND X[j] < X[i]}  se i > 1 e ∃j
  1                                         se non esiste tale j
}

Soluzione finale: max{LIS[1], LIS[2], ..., LIS[n]}
```

Algoritmo Bottom-Up

```
LIS_LENGTH(X, n)
for i = 1 to n:
  LIS[i] = 1      // Base: sequenza di un elemento
  prev[i] = nil   // Per ricostruzione
for i = 2 to n:
  for j = 1 to i-1:
    if X[j] < X[i] AND LIS[j] + 1 > LIS[i]:
      LIS[i] = LIS[j] + 1
```

```
      prev[i] = j
return max(LIS[1..n])
```

Complessità: $\Theta(n^2)$

Esempio Dettagliato: X = [8, 2, 5, 1, 3]

```
Calcolo LIS[i] per ogni posizione:

i=1: X[1]=8
  LIS[1] = 1 (sequenza: [8])

i=2: X[2]=2
  Controllo j=1: X[1]=8 > X[2]=2 → non crescente
  LIS[2] = 1 (sequenza: [2])

i=3: X[3]=5
  Controllo j=1: X[1]=8 > X[3]=5 → non crescente
  Controllo j=2: X[2]=2 < X[3]=5 → crescente!
    LIS[3] = LIS[2] + 1 = 2
  LIS[3] = 2 (sequenza: [2, 5])

i=4: X[4]=1
  Controllo j=1,2,3: tutti > 1
  LIS[4] = 1 (sequenza: [1])

i=5: X[5]=3
  Controllo j=1: 8 > 3 → no
  Controllo j=2: 2 < 3 → sì, LIS[5] = LIS[2] + 1 = 2
  Controllo j=3: 5 > 3 → no
  Controllo j=4: 1 < 3 → sì, LIS[5] = LIS[4] + 1 = 2
  LIS[5] = 2 (sequenze: [2,3] o [1,3])

Risultato: max(1, 1, 2, 1, 2) = 2

Possibili LIS: [2,5], [2,3], [1,3]
```

Dimostrazione Sottostruttura Ottima

```
Sia Z = LIS(Xᵢ) che termina in X[i]
Sia j < i tale che X[j] < X[i] e LIS[j] massimo

Allora Z = Z' ∪ {X[i]} dove Z' = LIS(Xⱼ)

Prova per assurdo:
- Supponi Z' non ottima per Xⱼ
- Esiste W migliore: |W| > |Z'|
```

```
- W ∪ {X[i]} sarebbe migliore di Z per Xᵢ
- Contraddizione con ottimalità di Z
```

PROBLEMA 3: SHORTEST PALINDROME COMPLETION (SPC)

Problema

Data stringa X[1..n], trova il **minimo numero di caratteri** da aggiungere per renderla palindroma.

Caratterizzazione Ricorsiva

```
SPC[i,j] = min caratteri da aggiungere a X[i..j] per renderla palindroma

SPC[i,j] = {
  0                            se i ≥ j (già palindroma)
  0                            se X[i] = X[j] e SPC[i+1,j-1] = 0
  SPC[i+1,j-1]                se X[i] = X[j]
  1 + min(SPC[i+1,j], SPC[i,j-1])  se X[i] ≠ X[j]
}
```

Algoritmo Bottom-Up

```
SPC_LENGTH(X, n)
for i = 1 to n: L[i,i] = 0
for i = 1 to n-1:
  if X[i] = X[i+1]:
    L[i,i+1] = 0
  else:
    L[i,i+1] = 1
for len = 3 to n:
  for i = 1 to n-len+1:
    j = i + len - 1
    if X[i] = X[j]:
      L[i,j] = L[i+1,j-1]
    else:
      L[i,j] = 1 + min(L[i+1,j], L[i,j-1])
return L[1,n]
```

Complessità: $\Theta(n^2)$

Esempio Dettagliato: X = "abc"

```
Tabella L[i,j]:

     a   b   c
  a  0   1   2
  b  -   0   1
  c  -   -   0

Calcolo:
- L[1,1] = L[2,2] = L[3,3] = 0 (singoli caratteri)
- L[1,2]: X[1]≠X[2] → L[1,2] = 1
- L[2,3]: X[2]≠X[3] → L[2,3] = 1
- L[1,3]: X[1]≠X[3] → L[1,3] = 1 + min(L[2,3], L[1,2]) = 1 + min(1,1) = 2

Risultato: 2 caratteri da aggiungere

Soluzioni possibili:
- "cbabc" (aggiungi "cb" all'inizio)
- "abcba" (aggiungi "ba" alla fine)
```

CONFRONTO: BOTTOM-UP vs MEMOIZATION

Aspetto	Bottom-Up	Memoization
Stile	Iterativo	Ricorsivo
Inizializzazione	Esplicita casi base	Check valore sentinella
Ordine calcolo	Fisso (dai più piccoli)	On-demand
Spazio	Sempre Θ(spazio tabella)	Variabile
Efficienza	Costante overhead	Overhead ricorsione
Debug	Più difficile	Più facile

Quando Usare Memoization?

- Quando non tutti i sottoproblemi servono
- Quando ordine di calcolo non ovvio
- Quando ricorsione è naturale per il problema

Quando Usare Bottom-Up?

- Quando tutti i sottoproblemi servono
- Quando serve massima efficienza
- Quando spazio è critico (si può ottimizzare)

OTTIMIZZAZIONI SPAZIO

Tecnica 1: Usa Solo Due Righe

Per problemi tipo LCS dove serve solo riga precedente:

```
// Invece di L[m+1][n+1]
curr[n+1]  // Riga corrente
prev[n+1]  // Riga precedente

Spazio: O(n) invece di O(m·n)
```

Tecnica 2: Calcolo Diagonale

Per SPC: calcola per lunghezza crescente

```
for len = 1 to n:  // Lunghezza sottostringa
  for i = 1 to n-len+1:
    j = i + len - 1
    // Calcola L[i,j]
```

PATTERN COMUNI PD

Pattern 1: Prefissi/Suffissi (LCS, Edit Distance)

- Sottoproblemi: tutti i prefissi
- Tabella: 2D, dimensione O(m·n)
- Riempimento: riga per riga o colonna per colonna

Pattern 2: Intervalli (SPC, Matrix Chain)

- Sottoproblemi: tutti gli intervalli [i,j]
- Tabella: 2D triangolare superiore
- Riempimento: per lunghezza crescente

Pattern 3: Posizione Finale (LIS, Subset Sum)

- Sottoproblemi: termina in posizione specifica
- Tabella: 1D, dimensione O(n)
- Riempimento: sinistra a destra

Pattern 4: Knapsack

- Sottoproblemi: primi i oggetti, capacità w
- Tabella: 2D, dimensione O(n·W)
- Riempimento: riga per riga

TEMPLATE RISOLUZIONE ESAME

Passo 1: Identificazione

✓ Problema di ottimizzazione?
✓ Sottoproblemi sovrapposti?
✓ Spazio sottoproblemi polinomiale?

Passo 2: Caratterizzazione

1. Definisci sottoproblema ottimale
2. Esprimi soluzione in funzione di sottoproblemi
3. Scrivi formula ricorsiva

Passo 3: Implementazione

```
1. Identifica spazio sottoproblemi
2. Alloca tabella
3. Inizializza casi base
4. Riempi tabella secondo ricorrenza
5. Return soluzione problema originale
```

Passo 4: Complessità

- Numero sottoproblemi × costo per sottoproblema
- Esempio: LCS ha O(mn) sottoproblemi, O(1) per sottoproblema → O(mn)

Passo 5: Esempio Numerico

- Input piccolo ma significativo
- Mostra tabella passo-passo
- Verifica risultato

CONFRONTO: PD vs GREEDY

Usa PD quando:

- Scelta ottima dipende da scelte future

- Più sottoproblemi da combinare
- Exchange argument non funziona
- Esempi: LCS, LIS, Knapsack 0/1

Usa Greedy quando:

- Scelta locale ovvia
- Scelta irrevocabile ma ottima
- Un solo sottoproblema dopo scelta
- Esempi: Activity Selection, Huffman, Dijkstra

ERRORI COMUNI DA EVITARE

❌ **Dimenticare casi base** → inizializza riga/colonna 0
❌ **Ordine riempimento sbagliato** → serve sottoproblemi già calcolati
❌ **Confondere i con j** → attento agli indici
❌ **Non gestire array 0-based vs 1-based**
❌ **Dimenticare ricostruzione soluzione** → non solo costo

✅ **Verifica sempre sottostruttura ottima**
✅ **Disegna tabella piccola a mano**
✅ **Test con input limite** (stringhe vuote, array singoletto)
✅ **Ottimizza spazio se possibile**

CHECKLIST ESAME PD

Prima di consegnare, verifica:

- [ ] Formula ricorsiva corretta
- [ ] Casi base gestiti
- [ ] Tabella dimensione corretta
- [ ] Ordine riempimento giusto
- [ ] Complessità calcolata (sottoproblemi × costo)
- [ ] Esempio numerico completo
- [ ] Ricostruzione soluzione (se richiesta)