📘 PROGRAMMAZIONE DINAMICA - TEMPLATE UNIVERSALE

Schema Generale

```
1. CARATTERIZZAZIONE RICORSIVA
   - Definisci struttura soluzione ottima S*
   - Esprimi S* in funzione di soluzioni ottime sottoproblemi S₁*, ..., Sₖ*

2. RELAZIONE DI RICORRENZA
   - Scrivi ricorrenza: cost(S) = f(cost(S₁*), ..., cost(Sₖ*))
   - Identifica casi base

3. CALCOLO BOTTOM-UP o MEMOIZATION
   - Bottom-up: iterativo, riempi tabella dal basso
   - Top-down: ricorsivo + tabella per memorizzare

4. RICOSTRUZIONE SOLUZIONE
   - (Opzionale) Mantieni info per ricostruire soluzione ottima
```

🔵 LCS - Longest Common Subsequence

Problema

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

Caratterizzazione

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

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

Pseudocodice 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
            else:
                L[i,j] = max(L[i-1,j], L[i,j-1])

    return L[m,n]
```

Complessità

- **Tempo:** Θ(mn)
- **Spazio:** O(mn) tabella completa, O(min(m,n)) con ottimizzazione

Esempio

```
X = "armo", Y = "oro"

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

LCS = 2 (sequenza: "ro")
```

🔵 LIS - Longest Increasing Subsequence

Problema (con Strengthening)

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

Perché Strengthening? Il problema originale LIS(X) non ha sottostruttura ottima diretta. Rafforziamo a: "LIS che termina in posizione i"

Caratterizzazione

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

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

Soluzione finale = max{LIS[i] : 1 ≤ i ≤ n}
```

Pseudocodice

```
LIS_Length(X, n):
    for i = 1 to n:
        L[i] = 1

    for i = 2 to n:
        for j = 1 to i-1:
            if X[j] < X[i]:
                L[i] = max(L[i], L[j] + 1)

    return max(L[1..n])
```

Complessità

- **Tempo:** $\Theta(n^2)$
- **Spazio:** $O(n)$

Esempio

```
X = [8, 2, 5, 1, 3]

L[1] = 1  (seq: [8])
L[2] = 1  (seq: [2])
L[3] = 2  (seq: [2,5])
L[4] = 1  (seq: [1])
L[5] = 2  (seq: [2,3])

max LIS = 2
```

🔵 Shortest Palindrome Completion (SPC)

Problema

Data stringa `X[1..n]`, trovare il numero minimo di caratteri da aggiungere alla fine per renderla palindroma.

Caratterizzazione

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

SPC[i,j] = {
    0                            se i ≥ j
    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]
}
```

Pseudocodice Bottom-Up

```
SPC(X, n):
    // Casi base
    for i = 1 to n:
        L[i,i] = 0
        if i < n:
            L[i,i+1] = (X[i] = X[i+1]) ? 0 : 1

    // Riempimento per lunghezze crescenti
    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

- **Tempo:** $\Theta(n^2)$
- **Spazio:** $O(n^2)$

🔵 Coin Change

Problema

Dato insieme monete `C = {c₁, ..., cₖ}` e target `T`, trovare il numero minimo di monete per ottenere T.

Caratterizzazione

```
MinCoins[i] = min numero di monete per ottenere somma i

MinCoins[i] = {
    0                                    se i = 0
    1 + min{MinCoins[i - cⱼ] : cⱼ ≤ i}  se i > 0
}
```

Pseudocodice

```
CoinChange(C, k, T):
    M[0] = 0
    for i = 1 to T:
        M[i] = ∞
        for j = 1 to k:
            if C[j] <= i:
                M[i] = min(M[i], 1 + M[i - C[j]])

    return M[T]
```

Complessità

- **Tempo:** Θ(Tk)
- **Spazio:** O(T)

⚡ ALGORITMI GREEDY - SCHEMA GENERALE

Due Proprietà Fondamentali

1. Proprietà Scelta Greedy Esiste sempre una soluzione ottima che contiene la scelta greedy. *Dimostrazione:* Exchange argument (cut & paste)

2. Sottostruttura Ottima Dopo la scelta greedy, il sottoproblema rimanente ha soluzione ottima.

Schema di Dimostrazione (Exchange Argument)

```
1. Sia A* una soluzione ottima qualunque
2. Sia a la scelta greedy
3. Se a ∈ A*, fine
4. Altrimenti, sostituisci un elemento di A* con a
5. Dimostra che la nuova soluzione è ancora ottima
6. Quindi esiste sempre una soluzione ottima che contiene la scelta greedy
```

🟢 Activity Selection

Problema

Date n attività con tempi inizio `s[i]` e fine `f[i]`, selezionare il massimo numero di attività compatibili (non sovrapposte).

Scelta Greedy

Seleziona l'attività che finisce per prima tra quelle rimanenti.

Dimostrazione Proprietà Scelta Greedy

```
Sia A* = {a₁, ..., aₖ} soluzione ottima con a₁ prima a finire in A*
Sia a la scelta greedy (finisce per prima in assoluto)

Se a = a₁ → ok
Se a ≠ a₁ → f[a] ≤ f[a₁] (per definizione greedy)
          → A' = {a, a₂, ..., aₖ} è ancora compatibile
          → |A'| = |A*| → A' è ottima e contiene scelta greedy
```

Pseudocodice

```
ActivitySelection(s, f, n):
    // Assumi f ordinato per tempo fine
    A = {1}
    j = 1
    for i = 2 to n:
        if s[i] >= f[j]:  // compatibile
            A = A ∪ {i}
            j = i
    return A
```

Complessità

- **Tempo:** O(n log n) se serve ordinare, O(n) se già ordinato
- **Spazio:** O(1)

Huffman Coding

Problema

Dato alfabeto con frequenze, costruire codice prefisso ottimale (lunghezza media minima).

Scelta Greedy

Unisci i due simboli con frequenza minima.

Algoritmo

```
Huffman(C):
    n = |C|
```

```
    Q = MinHeap(C)  // priority queue su frequenze

    for i = 1 to n-1:
        z = new Node()
        z.left = ExtractMin(Q)
        z.right = ExtractMin(Q)
        z.freq = z.left.freq + z.right.freq
        Insert(Q, z)

    return ExtractMin(Q)  // radice albero
```

Complessità

- **Tempo:** O(n log n)
- **Spazio:** O(n)

Esempio

```
Simboli: {A:12%, B:11%, C:53%, D:10%, E:14%}

1. Unisci D(10%) e B(11%) → DB(21%)
2. Unisci A(12%) e E(14%) → AE(26%)
3. Unisci DB(21%) e AE(26%) → DBAE(47%)
4. Unisci C(53%) e DBAE(47%) → radice(100%)

Codici risultanti:
C: 0
A: 110
E: 111
D: 100
B: 101
```

📊 CONFRONTO PD vs GREEDY

Aspetto	**Programmazione Dinamica**	**Algoritmi Greedy**
Scelte	Esamina tutte le possibilità	Una scelta per volta
Sottoproblemi	Multipli, sovrapposti	Singolo dopo scelta
Complessità	Spesso $O(n^2)$ o $O(n^3)$	Spesso O(n log n)
Garanzie	Sempre ottimo se possibile	Ottimo solo se proprietà greedy vale
Dimostrazione	Induzione + ricorrenza	Exchange argument
Esempi	LCS, LIS, SPC, Coin Change	Activity Selection, Huffman

---

🎯 CHECKLIST VELOCE

Usa Programmazione Dinamica se:

- [x] ~~Problema di ottimizzazione~~
- [x] ~~Sottostruttura ottima~~
- [x] ~~Sottoproblemi sovrapposti (stessa istanza calcolata più volte)~~
- [x] ~~Spazio sottoproblemi ragionevole~~

Usa Greedy se:

- [x] ~~Problema di ottimizzazione~~
- [x] ~~Sottostruttura ottima~~
- [x] ~~Proprietà scelta greedy dimostrabile~~
- [x] ~~Esiste criterio di ordinamento naturale~~
- [x] ~~Decisioni irrevocabili ma localmente ottime~~

💡 ERRORI COMUNI DA EVITARE

Programmazione Dinamica

- ❌ Non identificare tutti i casi base
- ❌ Sbagliare direzione riempimento tabella
- ❌ Non verificare sottostruttura ottima
- ❌ Dimenticare di ritornare max/min finale (non solo tabella)

Greedy

- ❌ Assumere greedy funziona sempre (es. Coin Change con {1, 20, 50})
- ❌ Non dimostrare proprietà scelta greedy
- ❌ Scegliere criterio greedy sbagliato (es. Activity: max durata invece di min fine)
- ❌ Dimenticare ordinamento iniziale quando necessario