Esercizio 1: Weighted Subsequence Sum

Dato array A[1..n] di interi, determinare la sottosequenza di somma massima tale che tra due elementi consecutivi della sottosequenza ci sia al più distanza k nell'array originale.

Caratterizzazione ricorrenza: Sia OPT(i) la somma massima di una sottosequenza che termina in posizione i.

```
OPT(i) = {
    A[i]                                              se i=1
    max{A[i], max{OPT(j) + A[i] : max(1,i-k) ≤ j < i}}   altrimenti
}
```

Algoritmo:

```
WEIGHTED_SUBSEQ(A, n, k)
    OPT[1] = A[1]
    for i = 2 to n
        OPT[i] = A[i]                  // costo c₁ (assegnazione)
        for j = max(1, i-k) to i-1     // al più k iterazioni
            if OPT[j] + A[i] > OPT[i]  // costo c₂ (confronto + somma)
                OPT[i] = OPT[j] + A[i] // costo c₃ (assegnazione)

    result = OPT[1]                    // costo c₄
    for i = 2 to n                     // n-1 iterazioni
        if OPT[i] > result             // costo c₅ (confronto)
            result = OPT[i]            // costo c₆ (assegnazione)
    return result
```

Analisi costo: Associando costo unitario ai confronti e alle somme, costo nullo alle altre operazioni:

$T(n) = \Sigma_{i=2}^{n} [c_2 \cdot \min(k, i-1)] + \Sigma_{i=2}^{n} c_5$

Nel caso peggiore ($k \geq n$): $T(n) = c_2 \cdot \Sigma_{i=2}^{n} (i-1) + c_5 \cdot (n-1) = c_2 \cdot \Sigma_{j=1}^{n-1} j + c_5 \cdot (n-1) = c_2 \cdot (n-1)n/2 + c_5 \cdot (n-1) = \Theta(n^2)$

Complessità: $\Theta(nk)$ tempo, $\Theta(n)$ spazio.

Esercizio 2: Matrix Bracket Optimization

Data sequenza di matrici $A_1 \times A_2 \times ... \times A_n$ con dimensioni $p_0 \times p_1$, $p_1 \times p_2$, ..., $p_{n-1} \times p_n$, determinare la parentesizzazione che minimizza il costo totale, dove moltiplicare due matrici i×j e j×k costa i·j·k operazioni scalari.

Caratterizzazione ricorrenza: Sia OPT(i,j) il costo minimo per calcolare $A_i A_{i+1} ... A_j$.

```
OPT(i,j) = {
    0                                                           se i=j
    min{OPT(i,k) + OPT(k+1,j) + pᵢ₋₁·pₖ·pⱼ : i ≤ k < j}      altrimenti
}
```

Algoritmo:

```
MATRIX_CHAIN(p, n)
    for i = 1 to n
        OPT[i][i] = 0                   // n assegnazioni

    for l = 2 to n                      // n-1 iterazioni (lunghezza
catena)
        for i = 1 to n-l+1              // n-l+1 iterazioni
            j = i + l - 1
            OPT[i][j] = ∞
            for k = i to j-1            // l-1 iterazioni
                cost = OPT[i][k] + OPT[k+1][j] + p[i-1]*p[k]*p[j]  // 1
moltiplicazione + 2 somme
                if cost < OPT[i][j]     // 1 confronto
                    OPT[i][j] = cost    // 1 assegnazione

    return OPT[1][n]
```

Analisi costo: Contando solo moltiplicazioni tra interi:

$T(n) = \Sigma_{l=2}^{n} \Sigma_{i=1}^{n-l+1} \Sigma_{k=i}^{j-1} 1 = \Sigma_{l=2}^{n} \Sigma_{i=1}^{n-l+1} (l-1)$

$= \Sigma_{l=2}^{n} (l-1)(n-l+1) = \Sigma_{m=1}^{n-1} m(n-m) = n \cdot \Sigma_{m=1}^{n-1} m - \Sigma_{m=1}^{n-1} m^2$

$= n \cdot (n-1)n/2 - (n-1)n(2n-1)/6 = n(n-1)/6 \cdot [3n - (2n-1)] = n(n-1)(n+1)/6$

$T(n) = (n^3-n)/6 = \Theta(n^3)$

Complessità: $\Theta(n^3)$ tempo, $\Theta(n^2)$ spazio.

Esercizio 3: Palindrome Partitioning con Costi

Data stringa S[1..n], determinare il numero minimo di tagli per dividere S in sottostringhe palindrome. Ogni taglio ha costo proporzionale alla lunghezza del prefisso tagliato.

Caratterizzazione ricorrenza: Sia OPT(i) il costo minimo per partizionare S[1..i] in palindromi.

```
OPT(i) = {
    0                                                         se i=0
    min{OPT(j-1) + j : 1 ≤ j ≤ i, S[j..i] è palindromo}     altrimenti
}
```

Algoritmo:

```
PALINDROME_PARTITION(S, n)
    // Precomputa tabella palindromi
    for i = 1 to n
        for j = i to n
            IS_PAL[i][j] = CHECK_PALINDROME(S, i, j)  // costo O(j-i+1) per
check

    OPT[0] = 0
    for i = 1 to n                       // n iterazioni
        OPT[i] = ∞
        for j = 1 to i                   // i iterazioni
            if IS_PAL[j][i]              // 1 confronto
                if OPT[j-1] + j < OPT[i] // 1 somma + 1 confronto
                    OPT[i] = OPT[j-1] + j // 1 somma + 1 assegnazione

    return OPT[n]

CHECK_PALINDROME(S, start, end)
    while start < end                    // al più (end-start+1)/2
iterazioni
        if S[start] ≠ S[end]            // 1 confronto caratteri
            return false
        start++; end--
    return true
```

Analisi costo: Contando confronti tra caratteri e operazioni aritmetiche:

Precomputazione palindromi: $T_1(n) = \Sigma_{i=1}^{n} \Sigma_{j=i}^{n} (j-i+1)/2 = \Sigma_{i=1}^{n} \Sigma_{k=1}^{n-i+1} k/2 = O(n^3)$

Ciclo principale: $T_2(n) = \Sigma_{i=1}^{n} i \cdot 2 = 2 \cdot \Sigma_{i=1}^{n} i = 2 \cdot n(n+1)/2 = n(n+1)$

$T(n) = O(n^3) + n(n+1) = O(n^3)$

Complessità: $\Theta(n^3)$ tempo, $\Theta(n^2)$ spazio.

Esercizio 4: Knapsack con Vincoli di Categoria

Dati n oggetti con peso w_i, valore v_i, categoria $c_i \in \{1,2,...,k\}$, e zaino di capacità W. Ogni categoria j può contribuire con al più M_j oggetti. Massimizzare il valore totale.

Caratterizzazione ricorrenza: Sia $OPT(i,w,m_1,...,m_k)$ il valore massimo usando primi i oggetti, peso ≤ w, e al più m_j oggetti dalla categoria j.

```
OPT(i,w,m₁,...,mₖ) = {
    0                                            se i=0 o w=0
    OPT(i-1,w,m₁,...,mₖ)                       se wᵢ > w o m꜀ᵢ = 0
    max{OPT(i-1,w,m₁,...,mₖ),                  altrimenti
        vᵢ + OPT(i-1,w-wᵢ,m₁,...,m꜀ᵢ-1,...,mₖ)}
}

dove cᵢ = categoria dell'oggetto i
```

Algoritmo (versione ottimizzata 3D):

```
CATEGORY_KNAPSACK(values, weights, categories, W, M, n, k)
    // OPT[w][j][count] = valore max con peso ≤ w, primi j oggetti di cat
corrente, count oggetti usati

    for w = 0 to W
        for j = 0 to n
            for count = 0 to max(M)
                if w = 0 or j = 0
                    OPT[w][j][count] = 0
                else
                    cat = categories[j]
                    OPT[w][j][count] = OPT[w][j-1][count]  // non prendo
oggetto j

                    if weights[j] ≤ w and count > 0        // 2 confronti
                        take = values[j] + OPT[w-weights[j]][j-1][count-1]
// 1 somma + 1 sottrazione
                        if take > OPT[w][j][count]          // 1 confronto
                            OPT[w][j][count] = take         // 1
```

```
assegnazione

    return OPT[W][n][M[cat_finale]]
```

Analisi costo: Contando operazioni aritmetiche e confronti:

$T(n) = W \cdot n \cdot \max(M) \cdot 4 = 4Wn \cdot \max(M)$

Complessità: $\Theta(Wn \cdot \max(M))$ tempo, $\Theta(Wn \cdot \max(M))$ spazio.

Esercizio 5: Edit Distance con Gap Penalties

Date stringhe X[1..m], Y[1..n], trovare l'allineamento di costo minimo con:

- Match/mismatch: costo 0 se X[i]=Y[j], δ altrimenti
- Gap di lunghezza l: $\alpha + \beta \cdot l$ (α apertura, β estensione)

Caratterizzazione ricorrenza: Tre matrici per tracciare lo stato corrente:

```
M[i][j] = min costo allineamento X[1..i], Y[1..j] terminando con
match/mismatch
I[i][j] = min costo allineamento X[1..i], Y[1..j] terminando con gap in X
D[i][j] = min costo allineamento X[1..i], Y[1..j] terminando con gap in Y

M[i][j] = {
    cost(X[i],Y[j]) + min{M[i-1][j-1], I[i-1][j-1], D[i-1][j-1]}  se i,j > 0
    ∞
altrimenti
}

I[i][j] = min{M[i-1][j] + α + β, I[i-1][j] + β, D[i-1][j] + α + β}
D[i][j] = min{M[i][j-1] + α + β, I[i][j-1] + α + β, D[i][j-1] + β}
```

Algoritmo:

```
GAP_ALIGNMENT(X, Y, m, n, α, β, δ)
    // Inizializzazione casi base
    M[0][j] = ∞, I[0][j] = ∞, D[0][j] = j·β + α  per j > 0
    M[i][0] = ∞, I[i][0] = i·β + α, D[i][0] = ∞  per i > 0

    for i = 1 to m                         // m iterazioni
        for j = 1 to n                     // n iterazioni
```

```
            // Calcola M[i][j]
            if X[i] = Y[j]               // 1 confronto caratteri
                match_cost = 0
            else
                match_cost = δ

            M[i][j] = match_cost + min3(M[i-1][j-1], I[i-1][j-1], D[i-1][j-
1])  // 2 somme, 2 confronti

            // Calcola I[i][j]
            I[i][j] = min3(M[i-1][j] + α + β, I[i-1][j] + β, D[i-1][j] + α +
β)  // 4 somme, 2 confronti

            // Calcola D[i][j]
            D[i][j] = min3(M[i][j-1] + α + β, I[i][j-1] + α + β, D[i][j-1] +
β)  // 4 somme, 2 confronti

    return min3(M[m][n], I[m][n], D[m][n])       // 2 confronti

min3(a, b, c)
    return min(min(a,b), c)                      // 2 confronti
```

Analisi costo: Contando operazioni aritmetiche e confronti:

$T(m,n) = mn \cdot (1 + 8 + 8 + 4) = 21mn$

Complessità: $\Theta(mn)$ tempo, $\Theta(mn)$ spazio.

Template Analisi Costo Esatta

Regole di calcolo:

1. **Operazioni unitarie**: confronti, somme, sottrazioni, moltiplicazioni tra interi
2. **Operazioni nulle**: assegnazioni, accessi array, chiamate funzione senza parametri
3. **Cicli**: costo = (numero iterazioni) × (costo corpo ciclo)
4. **Ricorrenze**: risolvere con sostituzione o Master Theorem

Somme utili:

- $\Sigma_{i=1}^{n} i = n(n+1)/2$
- $\Sigma_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$
- $\Sigma_{i=1}^{n} \Sigma_{j=i}^{n} 1 = n(n+1)/2$

- $\Sigma_{i=1}^{n} \Sigma_{j=1}^{i} 1 = n(n+1)/2$