FILE 6: ALGORITMI GREEDY

PARADIGMA GENERALE

Due Proprietà Fondamentali

1. Proprietà di Scelta Greedy

La soluzione ottima può essere costruita facendo scelte **localmente ottime** senza considerare conseguenze future.

Dimostrazione: Exchange Argument (Cut & Paste)

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

2. Sottostruttura Ottima

Dopo aver fatto la scelta greedy, il sottoproblema rimanente deve avere soluzione ottima.

Spazio sottoproblemi: tipicamente LINEARE (non quadratico come PD)

PROBLEMA 1: ACTIVITY SELECTION

Problema

Date n attività con:

- s[i] = tempo inizio attività i
- f[i] = tempo fine attività i

Seleziona il **massimo numero** di attività mutuamente compatibili (non sovrapposte).

Scelta Greedy

Seleziona sempre l'attività che finisce prima tra quelle disponibili.

Perché Funziona? (Exchange Argument)

```
Sia A* soluzione ottima qualunque
Sia a_m l'attività che finisce prima

Se a_m ∉ A*:
  Sia a_k ∈ A* l'attività che finisce prima in A*
  Sostituisco a_k con a_m:

  - f[a_m] ≤ f[a_k] (a_m finisce prima per definizione)
  - a_m compatibile con tutte le attività compatibili con a_k
  - |A*| rimane uguale → A* ancora ottima

Quindi esiste sempre soluzione ottima contenente scelta greedy ✓
```

Algoritmo

```
ACTIVITY_SELECTION(s, f, n)
  // PREREQUISITO: attività ordinate per f[i] crescente
1. A = {a[1]}           // Prima attività (finisce prima)
2. last = 1
3. for i = 2 to n:
4.   if s[i] ≥ f[last]:  // Compatibile con ultima scelta
5.     A = A ∪ {a[i]}
6.     last = i
7. return A
```

Complessità:

- O(n log n) per ordinamento iniziale
- O(n) per selezione
- **Totale: O(n log n)**

Esempio Completo

```
Input:
Attività:  a1  a2  a3  a4  a5  a6  a7
Inizio s:  1   4   2   3   7   8  11
Fine f:    3   6   9  10  11  12  13

Passo-passo:
1. Ordina per f: a1(1,3), a2(4,6), a3(2,9), a4(3,10), a5(7,11), a6(8,12),
a7(11,13)
2. Seleziona a1 (finisce a 3)
3. a2 inizia a 4 ≥ 3 → seleziona a2 (finisce a 6)
4. a3 inizia a 2 < 6 → salta
5. a4 inizia a 3 < 6 → salta
6. a5 inizia a 7 ≥ 6 → seleziona a5 (finisce a 11)
7. a6 inizia a 8 < 11 → salta
```

```
8. a7 inizia a 11 ≥ 11 → seleziona a7

Soluzione: {a1, a2, a5, a7} (4 attività)
```

Dimostrazione Sottostruttura Ottima

```
Sia A* soluzione ottima contenente scelta greedy a_m
Allora A* \ {a_m} è soluzione ottima per sottoproblema S_m
(attività che iniziano dopo fine di a_m)

Prova: per assurdo, se esistesse B* migliore per S_m:
  → B* ∪ {a_m} sarebbe migliore di A* per problema originale
  → Contraddizione con ottimalità di A*
```

PROBLEMA 2: HUFFMAN CODING

Problema

Dati caratteri con frequenze, costruisci codice prefisso che minimizza lunghezza media.

Scelta Greedy

Unisci sempre i due caratteri/nodi con frequenza minima.

Algoritmo

```
HUFFMAN(C)
  // C = insieme caratteri con frequenze
1. Q = MinHeap(C)  // Heap di priorità
2. for i = 1 to |C| - 1:
3.   x = ExtractMin(Q)
4.   y = ExtractMin(Q)
5.   z = CreateNode()
6.   z.freq = x.freq + y.freq
7.   z.left = x
8.   z.right = y
9.   Insert(Q, z)
10. return ExtractMin(Q)  // Radice albero
```

Complessità: O(n log n)

Esempio Dettagliato

```
Caratteri: {a:16, b:12, c:2, d:8, e:3, f:9, g:6}
```

```
Passo 1: Heap = [c:2, e:3, g:6, d:8, f:9, b:12, a:16]
  Merge c:2 + e:3 → z1:5
  Heap = [z1:5, g:6, d:8, f:9, b:12, a:16]

Passo 2: Merge z1:5 + g:6 → z2:11
  Heap = [d:8, f:9, z2:11, b:12, a:16]

Passo 3: Merge d:8 + f:9 → z3:17
  Heap = [z2:11, b:12, a:16, z3:17]

Passo 4: Merge z2:11 + b:12 → z4:23
  Heap = [a:16, z3:17, z4:23]

Passo 5: Merge a:16 + z3:17 → z5:33
  Heap = [z4:23, z5:33]

Passo 6: Merge z4:23 + z5:33 → root:56

Codifiche finali (0=sx, 1=dx):
c = 0000    (freq 2, len 4)
e = 0001    (freq 3, len 4)
g = 001     (freq 6, len 3)
d = 010     (freq 8, len 3)
f = 011     (freq 9, len 3)
b = 100     (freq 12, len 3)
a = 11      (freq 16, len 2)

Lunghezza media: (2×4 + 3×4 + 6×3 + 8×3 + 9×3 + 12×3 + 16×2) / 56 = 2.875
```

Proprietà Huffman

1. Caratteri con freq minore → codice più lungo
2. Albero binario completo (ogni nodo interno ha 2 figli)
3. Nessun codice è prefisso di un altro

PROBLEMA 3: INTERVAL COVERING

Problema

Dato insieme X = {x_1, x_2, ..., x_n} di punti ordinati su retta reale, trova insieme **minimo** di intervalli chiusi di ampiezza 1 che coprano tutti i punti.

Scelta Greedy

Copri il primo punto x_1 con intervallo [x_1, x_1+1], poi risolvi ricorsivamente.

Algoritmo

```
MIN_COVER(X, n)
1. last = 1
2. C = {[X[1], X[1] + 1]}  // Copre primo punto
3. for i = 2 to n:
4.   if X[i] > X[last] + 1:  // Fuori dall'ultimo intervallo
5.     last = i
6.     C = C ∪ {[X[i], X[i] + 1]}
7. return C
```

Complessità: O(n) (assumendo X ordinato)

Esempio

```
Input: X = {1, 1.5, 2.5, 3, 4, 4.5}

Soluzione:
1. Intervallo [1, 2] copre: 1, 1.5
2. Intervallo [2.5, 3.5] copre: 2.5, 3
3. Intervallo [4, 5] copre: 4, 4.5

Risultato: 3 intervalli
```

Dimostrazione Correttezza

```
Scelta greedy: [x₁, x₁+1]

Exchange argument:
Sia I* soluzione ottima con intervallo [a, a+1] che copre x₁
Se a = x₁ → fine
Se a < x₁:
  - Tutti i punti coperti da [a, a+1] sono anche coperti da [x₁, x₁+1]
  - Sostituisco [a, a+1] con [x₁, x₁+1]
  - |I*| rimane uguale → ancora ottima

Sottostruttura: dopo scelta greedy, I* \ {[x₁, x₁+1]} ottima per X \ {punti
coperti}
```

PROBLEMA 4: METRIC MATCHING ON THE LINE

Problema

Dati:

- $S = \{s_1, ..., s_n\}$ punti ordinati (server)
- $C = \{c_1, ..., c_n\}$ punti ordinati (client)

Assegna ogni client a server distinto minimizzando somma distanze $|c_i - s_j|$.

Scelta Greedy

Assegna primo client a primo server, secondo a secondo, ecc.

Algoritmo

```
METRIC_MATCHING(S, C, n)
1. for i = 1 to n:
2.   Match(C[i], S[i])
3. return matching
```

Complessità: O(n)

Esempio

```
Server S:  1   3   6   9
Client C:  2   4   7  10

Matching greedy:
C[1]=2 → S[1]=1, dist=1
C[2]=4 → S[2]=3, dist=1
C[3]=7 → S[3]=6, dist=1
C[4]=10 → S[4]=9, dist=1

Costo totale: 4
```

Perché Funziona?

```
Exchange argument:
Supponi matching ottimo con "incroci":
  c_i → s_j
  c_k → s_h  (con i < k e j > h)

Costo: |c_i − s_j| + |c_k − s_h|

Scambia:
  c_i → s_h
  c_k → s_j

Costo: |c_i − s_h| + |c_k − s_j|
```

```
Disuguaglianza triangolare garantisce: nuovo costo ≤ vecchio costo
Ripeti finché non ci sono più incroci → matching greedy
```

CONFRONTO: GREEDY vs PROGRAMMAZIONE DINAMICA

Aspetto	Greedy	PD
Scelte	Una per volta, irrevocabile	Esamina tutte le possibilità
Sottoproblemi	Singolo dopo scelta	Multipli, sovrapposti
Complessità	Spesso O(n log n)	Spesso $O(n^2)$ o $O(n^3)$
Garanzie	Ottimo solo se proprietà greedy vale	Sempre ottimo se possibile
Spazio	O(1) o O(n)	$O(n^2)$ tipico

QUANDO USARE GREEDY?

Checklist Diagnostica

✓ Esiste criterio di ordinamento naturale?
✓ Scelta locale sembra ovvia?
✓ Problema ha "flusso temporale"?
✓ Riesci a dimostrare exchange argument?

Problemi Tipici Greedy

1. **Activity Selection** → finire prima
2. **Huffman** → frequenza minima
3. **Minimum Spanning Tree** → arco peso minimo (Kruskal/Prim)
4. **Shortest Path** → distanza minima (Dijkstra)
5. **Fractional Knapsack** → rapporto valore/peso

Quando Greedy FALLISCE

```
Esempio: Cambio monete
Valori: [50, 20, 1]
Target: 60

Greedy: 50 + 1×10 = 11 monete
Ottimo: 20×3 = 3 monete
```

```
→ Greedy NON funziona per sistemi monetari arbitrari!
```

SCHEMA DI DIMOSTRAZIONE GREEDY

Passo 1: Proprietà Scelta Greedy (Exchange Argument)

```
1. Definisci scelta greedy
2. Considera soluzione ottima A* qualunque
3. Se scelta greedy ∈ A* → fine
4. Altrimenti: sostituisci elemento di A* con scelta greedy
5. Dimostra che nuova soluzione A' è ancora ottima
6. Conclusione: esiste sempre soluzione ottima con scelta greedy
```

Passo 2: Sottostruttura Ottima

```
1. Sia A* soluzione ottima contenente scelta greedy
2. Dopo rimozione scelta greedy, rimane sottoproblema S'
3. Dimostra: A* \ {scelta greedy} è ottima per S'
4. Metodo: dimostrazione per assurdo
   - Supponi ∃ B* migliore per S'
   - B* ∪ {scelta greedy} migliore di A* → contraddizione
```

Passo 3: Algoritmo

```
1. Ordina input secondo criterio greedy (se necessario)
2. Inizializza soluzione vuota
3. Per ogni elemento in ordine:
   - Se compatibile con scelte precedenti: aggiungi
4. Return soluzione costruita
```

Passo 4: Complessità

- Ordinamento: O(n log n)
- Selezione: O(n) o O(n log n) con heap
- Totale: tipicamente O(n log n)

TEMPLATE RISOLUZIONE ESAME

Per ogni problema greedy:

1. IDENTIFICAZIONE

- C'è criterio di ordinamento naturale?
- Scelta locale ovvia?

2. ALGORITMO

- Definisci scelta greedy
- Scrivi pseudocodice
- Calcola complessità

3. DIMOSTRAZIONE CORRETTEZZA

- Exchange argument per scelta greedy
- Sottostruttura ottima per sottoproblema

4. ESEMPIO NUMERICO

- Input piccolo ma significativo
- Mostra passo-passo le scelte
- Verifica ottimalità soluzione

ERRORI COMUNI DA EVITARE

❌ **Confondere greedy con euristica** → greedy è ottimo (se proprietà valgono)
❌ **Saltare dimostrazione** → sempre richiesta all'esame
❌ **Non verificare proprietà** → non tutti i problemi ammettono greedy
❌ **Errori nell'ordinamento** → cruciale per correttezza
❌ **Dimenticare casi base** → gestire input vuoto/singoletto

✅ **Verificare sempre proprietà scelta greedy**
✅ **Dimostrare sottostruttura ottima**
✅ **Testare con esempi limite**
✅ **Analizzare complessità formalmente**