Tutorato di Automi e Linguaggi Formali

Soluzioni Homework 11: NP-Completezza ed NP-Hard

Gabriel Rovesti
Corso di Laurea in Informatica – Università degli Studi di Padova

Tutorato 11 – 04-06-2025

1 Classi di Complessità P e NP

Esercizio 1. Considerare i seguenti problemi:

- $PATH = \{\langle G, s, t\rangle \mid G$ è un grafo che contiene un cammino da s a $t\}$
- $HAMPATH = \{\langle G, s, t\rangle \mid G$ è un grafo che contiene un cammino Hamiltoniano da s a $t\}$
- $RELPRIMES = \{\langle x, y\rangle \mid \gcd(x, y) = 1\}$

a) Dimostrare che $PATH \in P$ fornendo un algoritmo deterministico polinomiale esplicito e analizzandone la complessità.

b) Dimostrare che $HAMPATH \in NP$ costruendo un verificatore polinomiale. Specificare il formato del certificato e l'algoritmo di verifica.

c) Dimostrare che $RELPRIMES \in P$ utilizzando l'algoritmo di Euclide. Analizzare la complessità in termini della rappresentazione binaria degli input.

d) Spiegare perché $HAMPATH$ è considerato intrattabile mentre $PATH$ è trattabile, nonostante la loro apparente similarità.

Soluzione. **a) Dimostrazione che $PATH \in P$**

Teorema 1. *$PATH \in P$.*

Proof. Presentiamo un algoritmo deterministico polinomiale per $PATH$:

Listing 1: Algoritmo per PATH

```
def PATH(G, s, t):
    # Input: grafo G = (V, E), vertici s, t
    # Output: True se esiste cammino da s a t, False altrimenti

    visited = set()
    queue = [s]

    while queue:
        current = queue.pop(0)
        if current == t:
            return True
        if current not in visited:
            visited.add(current)
            for neighbor in G.neighbors(current):
                if neighbor not in visited:
                    queue.append(neighbor)

    return False
```

Analisi della complessità:

- Ogni vertice viene visitato al massimo una volta: $O(|V|)$
- Ogni arco viene esaminato al massimo una volta: $O(|E|)$
- Complessità totale: $O(|V| + |E|)$, che è polinomiale nella dimensione dell'input

Quindi $PATH \in P$. □

b) Dimostrazione che $HAMPATH \in NP$

Teorema 2. *$HAMPATH \in NP$.*

Proof. Costruiamo un verificatore polinomiale per $HAMPATH$:
Certificato: Una sequenza di vertici $c = v_1, v_2, \ldots, v_k$ dove $v_1 = s$ e $v_k = t$.
Verificatore: Su input $\langle\langle G, s, t\rangle, c\rangle$:

Listing 2: Verificatore per HAMPATH

```
def verify_HAMPATH(G, s, t, certificate):
    vertices = certificate.split(',')

    # Controlla che inizi con s e finisca con t
    if vertices[0] != s or vertices[-1] != t:
        return False

    # Controlla che tutti i vertici siano distinti
    if len(vertices) != len(set(vertices)):
```

```python
        return False

    # Controlla che contenga tutti i vertici del grafo
    if set(vertices) != set(G.vertices()):
        return False

    # Controlla che ci siano archi tra vertici consecutivi
    for i in range(len(vertices) - 1):
        if (vertices[i], vertices[i+1]) not in G.edges():
            return False

    return True
```

Analisi della complessità:

- Controllo inizio/fine: $O(1)$
- Controllo vertici distinti: $O(|V|)$
- Controllo copertura vertici: $O(|V|)$
- Controllo archi consecutivi: $O(|V|)$
- Complessità totale: $O(|V|)$, che è polinomiale

Quindi $HAMPATH \in NP$. □

c) Dimostrazione che $RELPRIMES \in P$

Teorema 3. *$RELPRIMES \in P$.*

Proof. Utilizziamo l'algoritmo di Euclide per calcolare $\gcd(x, y)$:

Listing 3: Algoritmo di Euclide

```python
def gcd(x, y):
    while y != 0:
        temp = y
        y = x % y
        x = temp
    return x

def RELPRIMES(x, y):
    return gcd(x, y) == 1
```

Analisi della complessità in rappresentazione binaria:

- Sia $n = \log x + \log y$ la dimensione dell'input
- Ad ogni iterazione, il resto $x \bmod y$ ha al massimo $\lfloor \log x \rfloor$ bit
- Il numero di iterazioni è al massimo $O(\log(\min(x, y))) = O(n)$

- Ogni operazione modulo richiede $O(n^2)$ tempo
- Complessità totale: $O(n^3)$, che è polinomiale nella dimensione dell'input

Quindi $RELPRIMES \in P$. □

d) Differenza tra $PATH$ e $HAMPATH$
La differenza fondamentale risiede nella natura del problema:

- **PATH**: Cerca qualsiasi cammino da s a t. L'algoritmo BFS/DFS può esplorare sistematicamente il grafo in modo efficiente.
- **HAMPATH**: Richiede un cammino che visiti ogni vertice esattamente una volta. Questo è un vincolo globale che richiede di considerare tutte le possibili permutazioni dei vertici.
- **Complessità strutturale**: PATH ha una struttura locale (basta seguire archi), mentre HAMPATH ha una struttura globale (deve soddisfare vincoli sull'intero grafo).
- **Spazio delle soluzioni**: PATH ha uno spazio di ricerca polinomiale, HAMPATH ha uno spazio esponenziale ($|V|!$ possibili ordinamenti).

Esercizio 2. Sia $DOMINO[2] = \{D \mid D$ è un insieme di tessere del domino tale che si possono disporre alcune tessere in fila in modo che ogni numero compaia esattamente due volte$\}$.

a) Dimostrare che $DOMINO[2] \in NP$ costruendo un verificatore polinomiale appropriato.

b) Confrontare la complessità di $DOMINO[2]$ con quella di $DOMINO[1]$ (disporre tutte le tessere in fila con numeri adiacenti corrispondenti). Spiegare perché $DOMINO[1] \in P$ mentre $DOMINO[2]$ è presumibilmente intrattabile.

c) Analizzare come piccole modifiche nella definizione di un problema possano cambiarne drasticamente la complessità computazionale.

Soluzione. **a) Dimostrazione che $DOMINO[2] \in NP$**

Teorema 4. *$DOMINO[2] \in NP$.*

Proof. Costruiamo un verificatore polinomiale per $DOMINO[2]$:
Certificato: Una sequenza di tessere $c = t_1, t_2, \ldots, t_k$ dove ogni t_i è una tessera da D.
Verificatore: Su input $\langle D, c\rangle$:

Listing 4: Verificatore per DOMINO[2

```
def verify_DOMINO2(D, certificate):
    tiles = certificate

```

```python
    # Controlla che tutte le tessere siano in D
    for tile in tiles:
        if tile not in D:
            return False

    # Controlla adiacenza: numeri consecutivi devono combaciare
    for i in range(len(tiles) - 1):
        if tiles[i].right != tiles[i+1].left:
            return False

    # Conta occorrenze di ogni numero
    number_count = {}
    for tile in tiles:
        number_count[tile.left] = number_count.get(tile.left, 0) + 1
        number_count[tile.right] = number_count.get(tile.right, 0) + 1

    # Controlla che ogni numero compaia esattamente 2 volte
    for count in number_count.values():
        if count != 2:
            return False

    return True
```

Analisi della complessità:

- Controllo appartenenza tessere: $O(k \cdot |D|)$ dove k è il numero di tessere nel certificato
- Controllo adiacenza: $O(k)$
- Conteggio numeri: $O(k)$
- Controllo condizione "esattamente 2": $O(\text{numero di numeri distinti})$
- Complessità totale: $O(k \cdot |D|)$, che è polinomiale

Quindi $DOMINO[2] \in NP$. □

b) Confronto tra $DOMINO[1]$ e $DOMINO[2]$

Teorema 5. *$DOMINO[1] \in P$ mentre $DOMINO[2]$ è presumibilmente intrattabile.*

Proof. **DOMINO[1] $\in P$:**

- $DOMINO[1]$ è riducibile al problema del cammino Euleriano
- Costruiamo un grafo dove i vertici sono i numeri sulle tessere e gli archi sono le tessere

- Un cammino Euleriano esiste se e solo se il grafo è connesso e ha al massimo 2 vertici di grado dispari
- L'algoritmo di Fleury risolve il problema in tempo $O(|E|^2) = O(|D|^2)$

DOMINO[2] è intrattabile:

- La condizione "ogni numero esattamente due volte" introduce un vincolo globale complesso
- Non esiste una riduzione ovvia a problemi polinomiali noti
- Il problema richiede di bilanciare simultaneamente:
  - Scelta di sottinsieme di tessere
  - Ordinamento delle tessere scelte
  - Vincoli di adiacenza
  - Vincoli di conteggio globale
- Lo spazio delle soluzioni è esponenziale nei casi generali

□

c) Impatto delle piccole modifiche

L'esempio di $DOMINO[1]$ vs $DOMINO[2]$ illustra come modifiche apparentemente minori possano causare salti drammatici nella complessità:

- **Struttura locale vs globale**: $DOMINO[1]$ ha vincoli solo locali (adiacenza), mentre $DOMINO[2]$ ha vincoli globali (conteggio).
- **Riducibilità**: $DOMINO[1]$ si riduce a problemi di teoria dei grafi ben compresi, $DOMINO[2]$ non ha riduzioni ovvie.
- **Algoritmi greedy**: $DOMINO[1]$ ammette strategie greedy (algoritmo di Fleury), $DOMINO[2]$ richiede considerazioni globali.
- **Fenomeno generale**: Questo è comune in complessità computazionale:
  - 2-SAT $\in P$, 3-SAT è NP-completo
  - 2-colorazione $\in P$, 3-colorazione è NP-completo
  - Matching $\in P$, 3-dimensional matching è NP-completo

2 Riduzioni Polinomiali

Esercizio 3. Dimostrare le seguenti riduzioni polinomiali fondamentali:

a) $CircuitSAT \leq_p SAT$: Costruire una riduzione che trasformi un circuito booleano in una formula proposizionale equivalente. Specificare la trasformazione per ogni tipo di porta logica (AND, OR, NOT).

b) $SAT \leq_p 3SAT$: Fornire un algoritmo che converta una formula booleana arbitraria in una formula in 3-CNF soddisfacibile se e solo se la formula originale è soddisfacibile.

c) Dimostrare che entrambe le riduzioni operano in tempo polinomiale, analizzando la crescita della dimensione dell'output rispetto all'input.

d) Spiegare l'importanza di queste riduzioni nella teoria della NP-completezza.

Soluzione. **a) Riduzione** $CircuitSAT \leq_p SAT$

Teorema 6. *$CircuitSAT \leq_p SAT$.*

Proof. Definiamo una riduzione che trasforma un circuito booleano C in una formula proposizionale equivalente ϕ.

Algoritmo di riduzione:

1. Per ogni porta del circuito, introduci una variabile che rappresenta il suo output
2. Per ogni porta, crea una formula che descrive la relazione tra input e output:
   - **AND**: Se $z = x \land y$, aggiungi $(z \leftrightarrow (x \land y))$
   - **OR**: Se $z = x \lor y$, aggiungi $(z \leftrightarrow (x \lor y))$
   - **NOT**: Se $z = \neg x$, aggiungi $(z \leftrightarrow \neg x)$
3. Combina tutte le formule in congiunzione e richiedi che l'output finale sia vero

Trasformazioni specifiche:

$$\begin{aligned} z = x \land y &\Rightarrow (z \to (x \land y)) \land ((x \land y) \to z) && (1) \\ &\equiv (\neg z \lor (x \land y)) \land (\neg(x \land y) \lor z) && (2) \\ &\equiv (\neg z \lor x) \land (\neg z \lor y) \land (\neg x \lor \neg y \lor z) && (3) \end{aligned}$$

$$z = x \lor y \Rightarrow (\neg z \lor x \lor y) \land (\neg x \lor z) \land (\neg y \lor z) \tag{4}$$

$$z = \neg x \Rightarrow (\neg z \lor \neg x) \land (x \lor z) \tag{5}$$

Il circuito è soddisfacibile se e solo se la formula risultante è soddisfacibile. □

b) Riduzione $SAT \leq_p 3SAT$

Teorema 7. *$SAT \leq_p 3SAT$.*

Proof. Convertiamo una formula arbitraria ϕ in 3-CNF attraverso i seguenti passi:

Passo 1: Conversione in CNF Usiamo le equivalenze logiche standard:

$$\neg(A \wedge B) \equiv \neg A \vee \neg B \tag{6}$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B \tag{7}$$
$$A \rightarrow B \equiv \neg A \vee B \tag{8}$$
$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A) \tag{9}$$

Passo 2: Conversione clausole in 3-CNF Per ogni clausola C:

- **Se** $|C| = 1$: $C = (l_1)$ diventa $(l_1 \vee y \vee z) \wedge (l_1 \vee y \vee \neg z) \wedge (l_1 \vee \neg y \vee z) \wedge (l_1 \vee \neg y \vee \neg z)$ dove y, z sono nuove variabili
- **Se** $|C| = 2$: $C = (l_1 \vee l_2)$ diventa $(l_1 \vee l_2 \vee y) \wedge (l_1 \vee l_2 \vee \neg y)$ dove y è una nuova variabile
- **Se** $|C| = 3$: C rimane invariata
- **Se** $|C| > 3$: $C = (l_1 \vee l_2 \vee \cdots \vee l_k)$ diventa:

$$(l_1 \vee l_2 \vee y_1) \wedge \tag{10}$$
$$(\neg y_1 \vee l_3 \vee y_2) \wedge \tag{11}$$
$$(\neg y_2 \vee l_4 \vee y_3) \wedge \tag{12}$$
$$\vdots \tag{13}$$
$$(\neg y_{k-3} \vee l_{k-1} \vee l_k) \tag{14}$$

dove $y_1, \ldots, y_{k-3}$ sono nuove variabili

La formula originale è soddisfacibile se e solo se la formula 3-CNF risultante è soddisfacibile. □

c) Analisi della complessità polinomiale

Teorema 8. *Entrambe le riduzioni operano in tempo polinomiale.*

Proof. **CircuitSAT $\leq_p$ SAT:**

- Ogni porta genera al massimo 3 clausole di dimensione al massimo 3
- Se il circuito ha n porte, la formula risultante ha $O(n)$ clausole
- La dimensione totale della formula è $O(n)$
- Tempo di costruzione: $O(n)$

SAT $\leq_p$ 3SAT:

- La conversione in CNF può aumentare la dimensione al massimo esponenzialmente, ma usando l'algoritmo di Tseitin rimane polinomiale
- Per ogni clausola di lunghezza k, generiamo $k-2$ nuove variabili e $k-2$ nuove clausole
- Se la formula originale ha m clausole con lunghezza totale ℓ, la formula 3-CNF ha $O(m+\ell)$ clausole
- Tempo di costruzione: $O(m+\ell)$

$\square$

d) Importanza nella teoria della NP-completezza
Queste riduzioni sono fondamentali perché:

1. **Catena di riduzioni**: Stabiliscono la catena $CircuitSAT \leq_p SAT \leq_p 3SAT$, che è alla base della teoria della NP-completezza
2. **Universalità di CircuitSAT**: La riduzione da $CircuitSAT$ mostra che ogni problema NP può essere ridotto a problemi di soddisfacibilità
3. **Forma normale standardizzata**: La riduzione a 3SAT fornisce una forma normale standardizzata per tutti i problemi NP-completi
4. **Base per ulteriori riduzioni**: 3SAT è spesso il punto di partenza per dimostrare la NP-completezza di altri problemi
5. **Teorema di Cook-Levin**: Queste riduzioni sono parte integrante della dimostrazione che 3SAT è NP-completo

Esercizio 4. Considerare la riduzione $3SAT \leq_p MAXINDSET$ (Insieme Indipendente Massimo).

a) Descrivere dettagliatamente la costruzione del grafo G a partire da una formula ϕ in 3-CNF. Specificare come vengono creati i vertici e gli archi.

b) Dimostrare che ϕ è soddisfacibile se e solo se G ha un insieme indipendente di dimensione m (dove m è il numero di clausole in ϕ).

c) Analizzare la complessità temporale della riduzione in termini del numero di variabili e clausole.

d) Utilizzare questa riduzione per concludere che $MAXINDSET$ è NP-completo.

Soluzione. a) Costruzione del grafo

Definizione 1. Data una formula 3-CNF $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ dove ogni $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$, costruiamo il grafo $G = (V, E)$ come segue:

Vertici: $V = \{v_{ij} \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$ Ogni vertice v_{ij} rappresenta il j-esimo letterale nella i-esima clausola.

Archi: $E = E_1 \cup E_2$ dove:

- $E_1 = \{(v_{ij}, v_{ik}) \mid 1 \leq i \leq m, 1 \leq j < k \leq 3\}$ (archi di clausola)
- $E_2 = \{(v_{ij}, v_{kl}) \mid i \neq k \text{ e } l_{ij} = \neg l_{kl}\}$ (archi di consistenza)

Esempio concreto: Se $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$:

- Vertici: $v_{11}(x_1), v_{12}(\neg x_2), v_{13}(x_3), v_{21}(\neg x_1), v_{22}(x_2), v_{23}(\neg x_3)$
- Archi di clausola: $(v_{11}, v_{12}), (v_{11}, v_{13}), (v_{12}, v_{13}), (v_{21}, v_{22}), (v_{21}, v_{23}), (v_{22}, v_{23})$
- Archi di consistenza: $(v_{11}, v_{21}), (v_{12}, v_{22}), (v_{13}, v_{23})$

b) Dimostrazione della correttezza

Teorema 9. *ϕ è soddisfacibile se e solo se G ha un insieme indipendente di dimensione m.*

Proof. ($\Rightarrow$) Supponiamo che ϕ sia soddisfacibile con assegnamento τ.

Per ogni clausola C_i, almeno un letterale l_{ij} è vero sotto τ. Scegliamo un tale letterale per ogni clausola e formiamo l'insieme $I = \{v_{ij} \mid l_{ij} \text{ è vero sotto } \tau \text{ e scelto per } C_i\}$.

Verifichiamo che I è un insieme indipendente:

- $|I| = m$ (un vertice per clausola)
- Non ci sono archi di clausola in I perché scegliamo al massimo un vertice per clausola
- Non ci sono archi di consistenza in I perché se $v_{ij}, v_{kl} \in I$ con $i \neq k$, allora l_{ij} e l_{kl} sono entrambi veri sotto τ, quindi non possono essere letterali contraddittori

($\Leftarrow$) Supponiamo che G abbia un insieme indipendente I di dimensione m.

Poiché $|I| = m$ e ci sono m clausole, I deve contenere esattamente un vertice per ogni clausola (altrimenti ci sarebbero archi di clausola in I).

Costruiamo un assegnamento τ come segue: per ogni variabile x, se esiste $v_{ij} \in I$ con $l_{ij} = x$, poni $\tau(x) = \text{vero}$; se esiste $v_{ij} \in I$ con $l_{ij} = \neg x$, poni $\tau(x) = \text{falso}$.

L'assegnamento è consistente perché non ci sono archi di consistenza in I. Ogni clausola è soddisfatta perché contiene almeno un letterale vero (quello corrispondente al vertice in I). $\square$

c) Analisi della complessità

Teorema 10. *La riduzione opera in tempo polinomiale.*

Proof. Sia n il numero di variabili e m il numero di clausole in ϕ.

Dimensione del grafo:

- Numero di vertici: $|V| = 3m$
- Numero di archi di clausola: $3m$ (3 archi per clausola)

- Numero di archi di consistenza: al massimo $O(m^2)$ (ogni coppia di vertici in clausole diverse)
- Numero totale di archi: $|E| = O(m^2)$

Tempo di costruzione:

- Creazione vertici: $O(m)$
- Creazione archi di clausola: $O(m)$
- Creazione archi di consistenza: $O(m^2)$ (controllo di tutti i possibili conflitti)
- Tempo totale: $O(m^2)$

Poiché tipicamente $m = O(n^k)$ per qualche costante k, la riduzione è polinomiale. □

d) NP-completezza di MAXINDSET

Teorema 11. *$MAXINDSET$ è NP-completo.*

Proof. **MAXINDSET ∈ NP:** Un certificato è un sottinsieme di vertici I. Il verificatore controlla in tempo polinomiale che:

- $|I| \geq k$
- Non ci sono archi tra vertici in I

MAXINDSET è NP-hard: Abbiamo dimostrato che $3SAT \leq_p MAXINDSET$. Poiché 3SAT è NP-completo, MAXINDSET è NP-hard.

Quindi MAXINDSET è NP-completo. □

3 Problemi NP-Completi su Grafi

Esercizio 5. Sia $VERTEXCOVER = \{\langle G, k\rangle \mid G$ ha una copertura di vertici di dimensione al più $k\}$.

a) Dimostrare che $VERTEXCOVER \in NP$ fornendo un verificatore polinomiale esplicito.

b) Dimostrare che $MAXINDSET \leq_p VERTEXCOVER$ utilizzando la relazione: I è un insieme indipendente in G se e solo se $V \setminus I$ è una copertura di vertici di G.

c) Concludere che $VERTEXCOVER$ è NP-completo utilizzando la riduzione precedente.

d) Confrontare questo risultato con il fatto che il problema della copertura di archi (maximum matching) è in P. Spiegare la differenza fondamentale.

Soluzione. **a) Dimostrazione che $VERTEXCOVER \in NP$**

Teorema 12. $VERTEXCOVER \in NP$.

Proof. Costruiamo un verificatore polinomiale per $VERTEXCOVER$:
Certificato: Un sottinsieme di vertici $C \subseteq V$.
Verificatore: Su input $\langle\langle G, k\rangle, C\rangle$:

Listing 5: Verificatore per VERTEXCOVER

```python
def verify_VERTEXCOVER(G, k, certificate):
    C = set(certificate)

    # Controlla che |C| <= k
    if len(C) > k:
        return False

    # Controlla che C sia una copertura di vertici
    for edge in G.edges():
        u, v = edge
        if u not in C and v not in C:
            return False  # Arco non coperto

    return True
```

Analisi della complessità:

- Controllo dimensione: $O(1)$
- Controllo copertura: $O(|E|)$
- Complessità totale: $O(|E|)$, che è polinomiale

Quindi $VERTEXCOVER \in NP$. □

b) Riduzione $MAXINDSET \leq_p VERTEXCOVER$

Teorema 13. $MAXINDSET \leq_p VERTEXCOVER$.

Proof. Definiamo la riduzione: data un'istanza $\langle G, k\rangle$ di MAXINDSET, costruiamo l'istanza $\langle G, |V| - k\rangle$ di VERTEXCOVER.

Lemma 1. *I è un insieme indipendente in G se e solo se $V \setminus I$ è una copertura di vertici di G.*

Dimostrazione del Lemma. (⇒) Sia I un insieme indipendente. Per ogni arco $\{u, v\} \in E$, poiché I è indipendente, al massimo uno tra u e v può essere in I. Quindi almeno uno tra u e v è in $V \setminus I$, rendendo $V \setminus I$ una copertura di vertici.

(⇐) Sia $V \setminus I$ una copertura di vertici. Se I non fosse indipendente, esisterebbe un arco $\{u, v\}$ con $u, v \in I$. Ma allora $u, v \notin V \setminus I$, contraddicendo il fatto che $V \setminus I$ copre tutti gli archi. □

Utilizzando il lemma: G ha un insieme indipendente di dimensione $\geq k$ se e solo se G ha una copertura di vertici di dimensione $\leq |V| - k$.

La riduzione è chiaramente polinomiale (tempo costante). □

c) NP-completezza di VERTEXCOVER

Teorema 14. *$VERTEXCOVER$ è NP-completo.*

Proof.
- $VERTEXCOVER \in NP$ (dimostrato in parte a)
- $MAXINDSET \leq_p VERTEXCOVER$ (dimostrato in parte b)
- Poiché MAXINDSET è NP-completo, VERTEXCOVER è NP-hard

Quindi VERTEXCOVER è NP-completo. □

d) Confronto con il maximum matching

La differenza fondamentale tra VERTEXCOVER (NP-completo) e maximum matching (in P) risiede nella struttura del problema:

- **Maximum Matching (in P):**
  - Cerca il massimo insieme di archi disgiunti
  - Ha struttura locale: ogni decisione (includere un arco) ha effetto solo sui vertici incidenti
  - Ammette algoritmi greedy e di programmazione lineare
  - Può essere risolto con l'algoritmo di Edmonds in $O(|V|^3)$
- **Vertex Cover (NP-completo):**
  - Cerca il minimo insieme di vertici che copre tutti gli archi
  - Ha struttura globale: ogni vertice può coprire molti archi, creando interdipendenze complesse
  - Non ammette algoritmi greedy ottimi
  - Richiede considerazione di tutte le combinazioni di vertici nel caso peggiore
- **Dualità interessante:**
  - Il matching massimo fornisce un lower bound per la vertex cover minima
  - Tuttavia, il gap può essere arbitrariamente grande
  - Questo illustra come problemi "duali" possano avere complessità molto diverse

Esercizio 6. Considerare il problema del circuito Hamiltoniano: $HAMCYCLE = \{\langle G \rangle \mid G \text{ ha un circuito Hamiltoniano}\}$.

a) Dimostrare che $HAMCYCLE \in NP$ costruendo un verificatore appropriato.

b) Spiegare perché $HAMCYCLE$ è NP-completo mentre il problema del circuito Euleriano è in P. Analizzare le differenze strutturali tra i due problemi.

c) Discutere l'algoritmo di Fleury per il circuito Euleriano e spiegare perché un approccio simile non funziona per il circuito Hamiltoniano.

d) Analizzare le implicazioni pratiche di questa differenza di complessità per problemi di routing e ottimizzazione.

Soluzione. a) Dimostrazione che $HAMCYCLE \in NP$

Teorema 15. *$HAMCYCLE \in NP$.*

Proof. Costruiamo un verificatore polinomiale per $HAMCYCLE$:
Certificato: Una sequenza di vertici $c = v_1, v_2, \ldots, v_n, v_1$ dove $n = |V|$.
Verificatore: Su input $\langle G, c\rangle$:

Listing 6: Verificatore per HAMCYCLE

```python
def verify_HAMCYCLE(G, certificate):
    vertices = certificate
    n = len(G.vertices())

    # Controlla che il ciclo abbia la lunghezza corretta
    if len(vertices) != n + 1:
        return False

    # Controlla che inizi e finisca con lo stesso vertice
    if vertices[0] != vertices[-1]:
        return False

    # Controlla che tutti i vertici del grafo siano visitati esattamente una volta
    visited = set(vertices[:-1])  # Escludi l'ultimo (duplicato del primo)
    if len(visited) != n or visited != set(G.vertices()):
        return False

    # Controlla che ci siano archi tra vertici consecutivi
    for i in range(len(vertices) - 1):
        if (vertices[i], vertices[i+1]) not in G.edges():
            return False

    return True
```

Analisi della complessità:

- Controlli di lunghezza e struttura: $O(n)$
- Controllo visitazione completa: $O(n)$
- Controllo archi consecutivi: $O(n)$
- Complessità totale: $O(n)$, che è polinomiale

Quindi $HAMCYCLE \in NP$. □

b) Differenze strutturali tra HAMCYCLE e circuito Euleriano

Teorema 16. *HAMCYCLE è NP-completo mentre il circuito Euleriano è in P.*

Analisi delle differenze. **Circuito Euleriano (in P):**

- **Definizione**: Visita ogni arco esattamente una volta
- **Caratterizzazione**: Esiste se e solo se il grafo è connesso e ogni vertice ha grado pari
- **Condizione locale**: Il grado di ogni vertice può essere controllato indipendentemente
- **Test di esistenza**: $O(|V| + |E|)$ per controllare connessione e gradi
- **Costruzione**: Algoritmo di Fleury in $O(|E|^2)$

Circuito Hamiltoniano (NP-completo):

- **Definizione**: Visita ogni vertice esattamente una volta
- **Caratterizzazione**: Non esiste una caratterizzazione semplice
- **Condizione globale**: Richiede considerazione dell'intera struttura del grafo
- **Test di esistenza**: Nessun algoritmo polinomiale noto
- **Costruzione**: Richiede ricerca esponenziale nello spazio delle permutazioni

Differenze chiave:

1. **Vincoli locali vs globali**: Eulero ha vincoli sui gradi (locali), Hamilton ha vincoli sulla struttura globale
2. **Flessibilità del percorso**: Eulero può ripetere vertici, Hamilton no
3. **ridondanza**: Eulero tollera percorsi alternativi, Hamilton richiede un percorso specifico

□

c) Algoritmo di Fleury e sue limitazioni
Perché Fleury non funziona per Hamilton:

- **Scelte irreversibili**: In Hamilton, una volta visitato un vertice, non può essere rivisitato. In Eulero, i vertici possono essere rivisitati.
- **Informazione locale vs globale**: Fleury usa informazione locale (ponti), ma Hamilton richiede informazione globale (rimanenti vertici da visitare).
- **Strategia greedy**: Fleury è greedy e funziona per Eulero, ma Hamilton non ammette strategie greedy ottime.

Algorithm 1 Algoritmo di Fleury per Circuito Euleriano

1. Inizia da un vertice arbitrario
2. A ogni passo, scegli un arco che:
   - Non sia un ponte (la cui rimozione disconnette il grafo)
   - Se tutti gli archi disponibili sono ponti, scegline uno arbitrario
3. Rimuovi l'arco scelto e continua dal vertice di destinazione
4. Ripeti fino a quando tutti gli archi sono stati percorsi

- **Backtracking necessario**: Hamilton spesso richiede backtracking, mentre Eulero può sempre procedere forward.

d) Implicazioni pratiche
Le differenze di complessità hanno impatti significativi:

- **Routing postale (Eulero)**:
  - Problema: Il postino deve percorrere ogni strada
  - Soluzione efficiente: Algoritmi polinomiali basati su matching
  - Applicazioni: Raccolta rifiuti, spazzamento strade, ispezione reti </itemize>
  - **Traveling Salesman (Hamilton)**:
    * Problema: Il venditore deve visitare ogni città
    * Nessuna soluzione efficiente esatta nota
    * Necessità di algoritmi approssimativi, euristiche, metodi probabilistici
    * Applicazioni: Logistica, pianificazione tour, routing veicoli
  - **Strategie pratiche per problemi Hamilton**:
    * Algoritmi approssimativi (es. Christofides per TSP)
    * Programmazione dinamica (Held-Karp $O(n^2 2^n)$)
    * Euristiche (nearest neighbor, 2-opt)
    * Metodi metaeuristici (simulated annealing, algoritmi genetici)
  - **Implicazioni economiche**:
    * I problemi Euleriani possono essere risolti ottimamente
    * I problemi Hamiltoniani richiedono trade-off tra tempo e qualità della soluzione
    * Necessità di investimenti in ricerca di algoritmi approssimativi

4 Strategie per Dimostrare NP-Completezza

Esercizio 7. Considerare il problema $3COLORING = \{\langle G\rangle \mid G$ è un grafo colorabile con 3 colori$\}$.

a) Seguire lo schema standard di dimostrazione di NP-completezza:
   - Dimostrare che $3COLORING \in NP$
   - Costruire una riduzione $3SAT \leq_p 3COLORING$
   - Provare la correttezza della riduzione
   - Analizzare la complessità temporale

b) Spiegare perché $2COLORING \in P$ mentre $3COLORING$ è NP-completo. Costruire un algoritmo polinomiale per $2COLORING$.

c) Discutere le strategie per scegliere il problema di partenza nelle riduzioni, utilizzando le linee guida presentate nella lezione.

Soluzione. **a) Dimostrazione di NP-completezza per 3COLORING**

Passo 1: $3COLORING \in NP$

Teorema 17. *$3COLORING \in NP$.*

Proof. **Certificato:** Una funzione di colorazione $c : V \to \{1, 2, 3\}$.

Verificatore: Su input $\langle G, c\rangle$:

Listing 7: Verificatore per 3COLORING

```
def verify_3COLORING(G, coloring):
    # Controlla che ogni vertice abbia un colore in {1, 2, 3}
    for vertex in G.vertices():
        if coloring[vertex] not in {1, 2, 3}:
            return False

    # Controlla che vertici adiacenti abbiano colori diversi
    for edge in G.edges():
        u, v = edge
        if coloring[u] == coloring[v]:
            return False

    return True
```

Complessità: $O(|V| + |E|)$, quindi $3COLORING \in NP$. □

Passo 2: Riduzione $3SAT \leq_p 3COLORING$

Teorema 18. *$3SAT \leq_p 3COLORING$.*

Proof. Data una formula 3-CNF $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ con variabili $x_1, \ldots, x_n$, costruiamo un grafo G colorabile con 3 colori se e solo se ϕ è soddisfacibile.

Costruzione del grafo:

1. Vertici base:

- Un vertice speciale T (rappresenta "vero")
- Per ogni variabile x_i: vertici x_i e $\neg x_i$
- Aggiungi arco $(x_i, \neg x_i)$ per ogni i
- Aggiungi archi (T, x_i) e $(T, \neg x_i)$ per ogni i

Questo crea un triangolo per ogni variabile: T, x_i, $\neg x_i$ devono avere colori diversi.

2. Gadget per clausole: Per ogni clausola $C_j = (l_{j1} \vee l_{j2} \vee l_{j3})$, aggiungiamo un gadget che forza almeno uno dei letterali ad essere "vero":

Listing 8: Costruzione gadget clausola

```
# Per clausola C_j = (l_j1 OR l_j2 OR l_j3)
# Crea vertici ausiliari
aux1_j, aux2_j, aux3_j, aux4_j, aux5_j = new_vertices()

# Struttura del gadget (OR a 3 input)
add_edges([
    (l_j1, aux1_j), (l_j2, aux1_j),    # Prima OR parziale
    (aux1_j, aux2_j), (l_j3, aux2_j),  # Seconda OR parziale
    (aux2_j, T),                       # Output deve essere diverso da T
    (aux1_j, aux3_j), (aux2_j, aux4_j), # Struttura di supporto
    (aux3_j, aux4_j), (aux3_j, aux5_j), (aux4_j, aux5_j), (T, aux5_j)
])
```

Il gadget garantisce che se tutti i letterali l_{j1}, l_{j2}, l_{j3} hanno lo stesso colore di T (cioè sono "falsi"), allora non è possibile colorare il gadget con 3 colori.

Passo 3: Correttezza della riduzione

Lemma 2. *ϕ è soddisfacibile se e solo se G è 3-colorabile.*

Proof. **(⇒)** Se ϕ è soddisfacibile con assegnamento τ:

- Colora T con colore 1
- Per ogni variabile x_i:
  - Se $\tau(x_i) =$ vero: colora x_i con 2, $\neg x_i$ con 3
  - Se $\tau(x_i) =$ falso: colora x_i con 3, $\neg x_i$ con 2
- Per ogni clausola C_j, almeno un letterale è "vero" (colore 2), quindi il gadget può essere colorato appropriatamente

(⇐) Se G è 3-colorabile:

- T deve avere un colore distinto (assumiamo 1)
- Per ogni variabile x_i, definisci $\tau(x_i) =$ vero se x_i ha colore 2
- Ogni gadget di clausola può essere colorato solo se almeno un letterale non ha colore 1, quindi almeno un letterale per clausola è vero

$\square$

Passo 4: Complessità temporale
La costruzione richiede:

- $O(n)$ vertici e archi per le variabili
- $O(1)$ vertici e archi per ogni clausola
- Tempo totale: $O(n+m)$, che è polinomiale

Quindi $3COLORING$ è NP-completo. $\square$

b) Differenza tra 2COLORING e 3COLORING

Teorema 19. *$2COLORING \in P$ mentre $3COLORING$ è NP-completo.*

Proof. **Algoritmo per 2COLORING:**

Listing 9: Algoritmo per 2COLORING

```
def two_coloring(G):
    color = {}

    for component in G.connected_components():
        if not component:
            continue

        # Prova a colorare la componente
        queue = [next(iter(component))]
        color[queue[0]] = 0

        while queue:
            current = queue.pop(0)
            current_color = color[current]

            for neighbor in G.neighbors(current):
                if neighbor in color:
                    # Controlla consistenza
                    if color[neighbor] == current_color:
                        return False  # Non bipartito
                else:
                    # Assegna colore opposto
                    color[neighbor] = 1 - current_color
                    queue.append(neighbor)

    return True
```

Analisi:

- Un grafo è 2-colorabile se e solo se è bipartito
- L'algoritmo fa BFS colorando ogni componente connessa

- Complessità: $O(|V| + |E|)$
- Quindi $2COLORING \in P$

Differenza fondamentale:

- **2-colorazione**: Equivale a verificare se il grafo è bipartito, proprietà controllabile localmente
- **3-colorazione**: Non esiste caratterizzazione semplice, richiede ricerca globale nello spazio delle colorazioni

$\square$

c) Strategie per scegliere il problema di partenza
Basandoci sulle linee guida della lezione:

- **Per 3COLORING**: Abbiamo usato 3SAT perché:
  - Il problema richiede di assegnare "etichette" (colori) agli oggetti (vertici)
  - Il numero 3 appare naturalmente in entrambi i problemi
  - 3SAT è molto versatile per costruzioni di gadget
- **Principi generali**:
  - **Assegnamento di bit**: Usa SAT o 3SAT
  - **Partizioni o etichettatura**: Usa 3SAT o 3COLORING
  - **Problemi di cammini/cicli**: Usa HAMPATH o HAMCYCLE
  - **Sottoinsiemi piccoli**: Usa VERTEXCOVER
  - **Sottoinsiemi grandi**: Usa MAXINDSET
  - **Quando il numero 3 è rilevante**: Prova 3SAT o 3COLORING
  - **In caso di dubbio**: 3SAT è la scelta più versatile
- **Considerazioni pratiche**:
  - Scegli il problema che condivide strutture simili
  - Considera la facilità di costruzione dei gadget
  - Preferisci problemi con dimostrazioni di NP-completezza ben note
  - Evita catene di riduzioni troppo lunghe

Esercizio 8. Definire il problema $SUBSETSUM = \{(S, t) \mid S$ è un insieme di interi positivi e esiste un sottoinsieme $S' \subseteq S$ tale che $\sum_{x \in S'} x = t\}$.

a) Dimostrare che $SUBSETSUM$ è NP-completo costruendo una riduzione appropriata da $3SAT$.

b) Nella costruzione della riduzione, spiegare come:
   - Rappresentare variabili booleane come interi
   - Codificare clausole nella somma target
   - Garantire che la riduzione preservi soddisfacibilità

c) Analizzare l'esistenza di algoritmi pseudo-polinomiali per $SUBSETSUM$ e discutere la diferenza tra complesità forte e debole.

d) Confrontare con il problema della somma esatta vs. il problema della somma approssimata.

Soluzione. a) Dimostrazione che SUBSETSUM è NP-completo
Passo 1: $SUBSETSUM \in NP$

Teorema 20. $SUBSETSUM \in NP$.

Proof. **Certificato:** Un sottinsieme $S' \subseteq S$.
Verificatore: Calcola $\sum_{x \in S'} x$ e verifica se uguale a t. Complessità: $O(|S'|) = O(|S|)$, quindi $SUBSETSUM \in NP$. □

Passo 2: Riduzione $3SAT \leq_p SUBSETSUM$

Teorema 21. $3SAT \leq_p SUBSETSUM$.

Proof. Data una formula 3-CNF ϕ con n variabili $x_1, \ldots, x_n$ e m clausole $C_1, \ldots, C_m$, costruiamo un'istanza (S, t) di SUBSETSUM.
Rappresentazione numerica: Usiamo un sistema posizionale in base 10 con $n+m$ cifre:

- Posizioni $1, \ldots, n$: corrispondono alle variabili $x_1, \ldots, x_n$
- Posizioni $n+1, \ldots, n+m$: corrispondono alle clausole $C_1, \ldots, C_m$

Costruzione dell'insieme S**:**
1. Numeri per le variabili: Per ogni variabile x_i $(1 \leq i \leq n)$:

- Numero v_i: ha cifra 1 in posizione i, e cifra 1 in posizione $n+j$ se x_i appare in clausola C_j
- Numero $\neg v_i$: ha cifra 1 in posizione i, e cifra 1 in posizione $n+j$ se $\neg x_i$ appare in clausola C_j

2. Numeri slack per le clausole: Per ogni clausola C_j $(1 \leq j \leq m)$:

- Numero s_{j1}: ha cifra 1 solo in posizione $n+j$
- Numero s_{j2}: ha cifra 1 solo in posizione $n+j$

Numero target t**:** t ha cifra 1 in posizioni $1, \ldots, n$ (una scelta per variabile) e cifra 3 in posizioni $n+1, \ldots, n+m$ (tre letterali per clausola).
Esempio concreto: Per $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$:

Numero	x_1	x_2	x_3	C_1	C_2
v_1	1	0	0	1	0
$\neg v_1$	1	0	0	0	1
v_2	0	1	0	0	1
$\neg v_2$	0	1	0	1	0
v_3	0	0	1	1	0
$\neg v_3$	0	0	1	0	1
s_{11}	0	0	0	1	0
s_{12}	0	0	0	1	0
s_{21}	0	0	0	0	1
s_{22}	0	0	0	0	1
Target	1	1	1	3	3

Correttezza della riduzione:

Lemma 3. *ϕ è soddisfacibile se e solo se (S, t) ha soluzione.*

Proof. ($\Rightarrow$) Se ϕ è soddisfacibile con assegnamento τ:

- Per ogni x_i: includi v_i se $\tau(x_i) =$ vero, altrimenti $\neg v_i$
- Questo dà cifra 1 in ogni posizione variabile
- Per ogni clausola C_j: almeno un letterale è vero, quindi la somma nelle posizioni clausola è ≥ 1
- Usa i numeri slack s_{ji} per portare ogni posizione clausola esattamente a 3

($\Leftarrow$) Se (S, t) ha soluzione S':

- Per ogni variabile x_i: esattamente uno tra v_i e $\neg v_i$ è in S' (per ottenere cifra 1)
- Defisci $\tau(x_i) =$ vero se $v_i \in S'$
- Per ogni clausola C_j: per ottenere cifra 3, servono contributi da variabili + slack
- Se nessun letterale di C_j fosse vero, la somma sarebbe < 3 anche con tutti i slack

$\square$

b) Dettagli della costruzione
Rappresentazione variabili:

- Ogni variabile booleana diventa una "scelta" tra due numeri
- La scelta di v_i vs $\neg v_i$ corrisponde a $x_i =$ vero vs $x_i =$ falso
- Il vincolo di somma forza esattamente una scelta per variabile

Codifica clausole:

- Ogni clausola contribuisce alla somma attraverso i suoi letterali
- Il target 3 per clausola garantisce che almeno un letterale sia scelto
- I numeri slack permettono flessibilità nel raggiungere esattamente 3

Preservazione soddisfacibilità:

- La costruzione è biettiva: ogni assegnamento corrisponde a un sottinsieme
- I vincoli numerici replicano esattamente i vincoli logici della formula

c) Algoritmi pseudo-polinomiali

Teorema 22. *SUBSETSUM ammette un algoritmo pseudo-polinomiale di programmazione dinamica.*

Proof.

Listing 10: Algoritmo DP per SUBSETSUM

```
def subset_sum_dp(S, t):
    n = len(S)
    # dp[i][j] = True se usando primi i elementi possiamo
       ottenere somma j
    dp = [[False] * (t + 1) for _ in range(n + 1)]
```

```python

    # Caso base: somma 0 sempre ottenibile (sottinsieme vuoto
        )
    for i in range(n + 1):
        dp[i][0] = True

    for i in range(1, n + 1):
        for j in range(t + 1):
            # Non includere S[i-1]
            dp[i][j] = dp[i-1][j]

            # Includere S[i-1] se possibile
            if j >= S[i-1]:
                dp[i][j] = dp[i][j] or dp[i-1][j - S[i-1]]

    return dp[n][t]
```

Analisi della complessità:

- Tempo: $O(n \cdot t)$ dove $n = |S|$ e t è il target
- Spazio: $O(n \cdot t)$ (ottimizzabile a $O(t)$)
- **Pseudo-polinomiale**: polinomiale in n e nel *valore* di t, ma esponenziale nella *rappresentazione* di t

□

Complessità forte vs debole:

Definizione 2. Un problema è:

- **Debolmente NP-completo**: NP-completo ma ammette algoritmi pseudo-polinomiali
- **Fortemente NP-completo**: NP-completo anche quando tutti i numeri sono limitati polinomialmente

SUBSETSUM è debolmente NP-completo:

- Se $t = O(\text{poly}(n))$, allora l'algoritmo DP è polinomiale
- Il problema diventa difficile solo quando t è esponenzialmente grande
- Contrasto con 3PARTITION, che è fortemente NP-completo

d) Somma esatta vs approssimata

- **Somma esatta (SUBSETSUM)**: NP-completo, ma ammette:
  - Algoritmi pseudo-polinomiali
  - FPTAS (Fully Polynomial-Time Approximation Scheme)
  - Algoritmi pratici per istanze di dimensioni moderate
- **Somma approssimata**: Versioni del problema più trattabili:
  - "Esiste un sottinsieme con somma $\geq (1 - \epsilon)t$?" è più facile

– Algoritmi di approssimazione con garanzie teoriche
– Utile per applicazioni pratiche dove la precisione esatta non è critica

- **Applicazioni pratiche**:
  – Problemi di zaino: spesso la soluzione approssimata è sufficiente
  – Partizionamento di risorse: tolleranza per piccoli sbilanciamenti
  – Crittografia: la versione esatta è necessaria per la sicurezza

5 Problemi Avanzati di NP-Completezza

Esercizio 9. Definire il problema $PARTITION = \{S \mid S$ è un insieme di interi positivi che può essere partizionato in due sottoinsiemi con la stessa somma$\}$.

a) Dimostrare che $PARTITION$ è NP-completo riducendo da $SUBSETSUM$.
b) Analizzare la relazione tra $PARTITION$ e altri problemi classici di programmazione dinamica.
c) Discutere l'esistenza di algoritmi pseudo-polinomiali per $PARTITION$ e le implicazioni per la complessità parametrizzata.
d) Estendere l'analisi al problema $3PARTITION$ e discutere perché è fortemente NP-completo.

Soluzione. a) Dimostrazione che PARTITION è NP-completo
Passo 1: $PARTITION \in NP$

Teorema 23. $PARTITION \in NP$.

Proof. **Certificato:** Un sottinsieme $S_1 \subseteq S$.
Verificatore: Su input $\langle S, S_1 \rangle$:

1. Calcola $sum_1 = \sum_{x \in S_1} x$
2. Calcola $sum_2 = \sum_{x \in S \setminus S_1} x$
3. Accetta se $sum_1 = sum_2$

Complessità: $O(|S|)$, quindi $PARTITION \in NP$. □

Passo 2: Riduzione $SUBSETSUM \leq_p PARTITION$

Teorema 24. $SUBSETSUM \leq_p PARTITION$.

Proof. Data un'istanza (S, t) di SUBSETSUM, costruiamo un'istanza S' di PARTITION.
Costruzione:

1. Calcola $\sigma = \sum_{x \in S} x$ (somma totale)
2. Se $t > \sigma$ o $t \leq 0$, restituisci un'istanza banale di PARTITION (es. $\{1, 3\}$ - non partizionabile)
3. Altrimenti, costruisci $S' = S \cup \{2\sigma - t, \sigma + t\}$

Analisi della correttezza:
La somma totale di S' è:

$$\sigma + (2\sigma - t) + (\sigma + t) = 4\sigma$$

Per una partizione bilanciata, ogni parte deve avere somma 2σ.

Lemma 4. *(S, t) ha soluzione per SUBSETSUM se e solo se S' ha una partizione bilanciata.*

Proof. **(⇒)** Se esiste $T \subseteq S$ con $\sum_{x \in T} x = t$:
Costruiamo la partizione:

- $S_1 = T \cup \{2\sigma - t\}$
- $S_2 = (S \setminus T) \cup \{\sigma + t\}$

Verifichiamo le somme:

$$\sum_{x \in S_1} x = t + (2\sigma - t) = 2\sigma \tag{15}$$

$$\sum_{x \in S_2} x = (\sigma - t) + (\sigma + t) = 2\sigma \tag{16}$$

(⇐) Se S' ha una partizione bilanciata S_1, S_2:
I due nuovi elementi $\{2\sigma - t, \sigma + t\}$ non possono stare nella stessa parte perché:

$$(2\sigma - t) + (\sigma + t) = 3\sigma > 2\sigma$$

Assumiamo $2\sigma - t \in S_1$ e $\sigma + t \in S_2$.
Sia $T = S_1 \cap S$ (parte di S in S_1). Allora:

$$\sum_{x \in T} x + (2\sigma - t) = 2\sigma$$

$$\Rightarrow \sum_{x \in T} x = t$$

Quindi T è soluzione per SUBSETSUM. □

La riduzione opera chiaramente in tempo polinomiale. □

b) Relazione con altri problemi di programmazione dinamica
PARTITION è strettamente correlato a diversi problemi classici:

- **Knapsack 0/1**:
  - PARTITION è un caso speciale dove capacità = valore = metà della somma totale
  - Entrambi ammettono soluzioni DP pseudo-polinomiali
  - La struttura della ricorrenza è identica
- **SUBSETSUM**:
  - PARTITION equivale a SUBSETSUM con target = metà della somma

- Entrambi debolmente NP-completi
- Stesse tecniche algoritmiche applicabili

- **Bin Packing**:
  - PARTITION è bin packing con 2 bin di capacità illimitata
  - Generalizzazioni naturali verso problemi di scheduling
  - Tecniche di approssimazione simili

c) Algoritmi pseudo-polinomiali per PARTITION

Teorema 25. *PARTITION ammette un algoritmo pseudo-polinomiale.*

Proof.

Listing 11: Algoritmo DP per PARTITION

```python
def partition_dp(S):
    total_sum = sum(S)

    # Se la somma  dispari, impossibile partizionare
    if total_sum % 2 != 0:
        return False

    target = total_sum // 2
    n = len(S)

    # dp[i][j] = True se usando primi i elementi possiamo
       ottenere somma j
    dp = [[False] * (target + 1) for _ in range(n + 1)]

    # Caso base
    for i in range(n + 1):
        dp[i][0] = True

    for i in range(1, n + 1):
        for j in range(target + 1):
            dp[i][j] = dp[i-1][j]  # Non includere S[i-1]

            if j >= S[i-1]:
                dp[i][j] = dp[i][j] or dp[i-1][j - S[i-1]]

    return dp[n][target]
```

Complessità:

- Tempo: $O(n \cdot \sum S)$ - pseudo-polinomiale
- Spazio: $O(n \cdot \sum S)$ (ottimizzabile a $O(\sum S)$)

$\square$

Implicazioni per la complessità parametrizzata:

- **Parametro somma totale**: Se $\sum S = O(\text{poly}(n))$, PARTITION $\in$ P
- **Parametro numero di elementi**: Problema rimane NP-completo anche per n piccolo se i numeri sono grandi
- **Fixed-Parameter Tractability**: PARTITION è FPT rispetto alla somma massima degli elementi
- **Kernel**: Esiste un kernel polinomiale per certi parametri

d) Estensione a 3PARTITION

Definizione 3. $3PARTITION = \{S \mid |S| = 3m, \sum S = mB$ e S può essere partizionato in m triple, ognuna con somma $B\}$ dove ogni elemento $s \in S$ soddisfa $B/4 < s < B/2$.

Teorema 26. *3PARTITION è fortemente NP-completo.*

Sketch. **Differenze chiave da PARTITION:**

- **Vincoli rigidi**: Ogni tripla deve avere esattamente somma B
- **Dimensioni degli elementi**: $B/4 < s < B/2$ implica che ogni tripla contiene esattamente 3 elementi
- **Nessuna flessibilità**: Non si può "bilanciare" tra diverse parti

Forte NP-completezza:

- Il problema rimane NP-completo anche quando tutti i numeri sono limitati polinomialmente
- Non esiste algoritmo pseudo-polinomiale (assumendo P NP)
- Anche con $B = O(\text{poly}(m))$, il problema è intrattabile

Riduzione da 3SAT: La costruzione è più sofisticata di PARTITION, ma la chiave è che:

- Ogni "scelta" booleana si traduce in esattamente una configurazione di triple
- I vincoli rigidi di 3PARTITION prevengono soluzioni parziali
- La rigidità strutturale rende il problema intrattabilmente difficile

□

Conseguenze pratiche:

- **PARTITION**: Problemi pratici risolvibili con DP per dimensioni moderate
- **3PARTITION**: Richiede sempre tecniche approssimative o euristiche
- **Applicazioni**: Scheduling rigido, allocazione risorse con vincoli stretti
- **Approssimazione**: 3PARTITION non ammette PTAS (assumendo P NP)

6 Conclusioni

La teoria della NP-completezza fornisce un framework fondamentale per comprendere i limiti computazionali e guidare la progettazione di algoritmi efficienti. Attraverso gli esercizi risolti, abbiamo visto come:

- Le riduzioni polinomiali stabiliscono relazioni precise tra problemi computazionali
- Piccole modifiche nella definizione di un problema possono causare salti drammatici nella complessità
- La distinzione tra complessità forte e debole ha implicazioni pratiche significative
- Le tecniche di dimostrazione di NP-completezza seguono schemi riconoscibili e applicabili

Questi risultati continuano a influenzare lo sviluppo di algoritmi approssimativi, euristiche e metodi per affrontare problemi computazionalmente intrattabili nella pratica.