#: 1F
Vertici: Ordini
Archi (condizione): Sequenza cronologica, ma solo se quantità totale ≥ Q
Peso arco: Giorni di distanza
Input utente: Q
Possibili domande:
• Path più breve in termini di tempo
• Individua eventuali cicli (dovrebbe essere DAG)
#: 2F
Tema: Catena di ordini con sconto
Vincoli ricorsivi: Ogni arco deve avere discount maggiore del precedente
• max 20 ordini
Obiettivo da massimizzare: Sconto medio del percorso

1F – “Grafo di ordini cronologico”

Input utente

Q = quantità minima totale di ciascun ordine

N = distanza massima in giorni fra due ordini

1F.1 Vertici

Ogni ordine con somma totale delle quantità ≥ Q:

```
SELECT
    o.order_id,
    o.order_date
FROM orders o,
     order_items oi
WHERE o.order_id = oi.order_id
GROUP BY o.order_id, o.order_date
HAVING SUM(oi.quantity) >= %s;
```

Parametro: (Q,)

1F.2 Archi

Arco orientato o1 → o2 se

1. o2 è cronologicamente successivo a o1,

2. distanza in giorni fra le date ≤ N,

3. entrambe gli ordini soddisfano la soglia Q.

```
SELECT
   v1.order_id        AS o1,
   v2.order_id        AS o2,
   DATEDIFF(v2.order_date, v1.order_date) AS days_diff
FROM
  (
   SELECT o.order_id, o.order_date
   FROM orders o, order_items oi
   WHERE o.order_id = oi.order_id
   GROUP BY o.order_id, o.order_date
   HAVING SUM(oi.quantity) >= %s
  ) v1,
  (
   SELECT o.order_id, o.order_date
   FROM orders o, order_items oi
   WHERE o.order_id = oi.order_id
   GROUP BY o.order_id, o.order_date
   HAVING SUM(oi.quantity) >= %s
  ) v2
WHERE v1.order_date < v2.order_date
  AND DATEDIFF(v2.order_date, v1.order_date) <= %s;
```

Parametri (binding): (Q, Q, N)

1F.3 Python: domande sul grafo

```
import networkx as nx

def build_order_graph(dao, Q, N):
    G = nx.DiGraph()
    # 1) Vertici
    verts = dao.get_orders(Q)          # [(order_id, order_date), ...]
    G.add_nodes_from(o for o,_ in verts)
    # 2) Archi
    for o1, o2, days in dao.get_order_edges(Q, N):
        G.add_edge(o1, o2, weight=days)
    return G

def shortest_time_path(G, start, end):
    """
    Cammino di peso minimo (somma di days_diff).
    """
    return nx.shortest_path(G, start, end, weight='weight')

def find_cycles(G):
    """
    Ritorna un ciclo se esiste, altrimenti lista vuota.
```

```
    """
    try:
        return nx.find_cycle(G, orientation='original')
    except nx.NetworkXNoCycle:
        return []
```

dao.get_orders(Q) esegue la query 1F.1

dao.get_order_edges(Q, N) esegue la query 1F.2

2F – “Catena di ordini con sconto crescente”

Vincoli ricorsivi

Cammino semplice (niente ripetizioni)

Ogni arco $o_i \rightarrow o_{i+1}$ deve avere discount(o_{i+1}) > discount(o_i)

Max 20 ordini

Obiettivo

Massimizzare lo **sconto medio** sul percorso.

2F.1 Funzione madre (getBestChain)

```
class OrderDiscountChain:
    def __init__(self, graph):
        self.G = graph
        self.best_chain = []
        self.best_score = 0.0

    def getBestChain(self, start):
        """
        Avvia il backtracking partendo dal nodo `start`.
        """
        self.best_chain = []
        self.best_score = 0.0
        for nbr in self.G.neighbors(start):
            # primo arco deve già soddisfare sconto crescente
            if self.G[nbr][start]['discount'] > self.G[start][start]['discount']:
                self._ricorsione([start, nbr])
        return self.best_chain, self.best_score
```

Nota: se discount è un attributo di **nodo** anziché di **arco**, adatta il confronto a self.G.nodes[...].

2F.2 Funzione ricorsiva (_ricorsione)

```
def _ricorsione(self, chain):
    # 1) calcola lo sconto medio corrente
    score = self.getScore(chain)
    if score > self.best_score:
        self.best_score = score
        self.best_chain = chain.copy()

    # 2) stop se raggiunti 20 ordini
    if len(chain) >= 20:
        return

    prev, last = chain[-2], chain[-1]
    prev_disc = self.G[prev][last]['discount']

    # 3) esplora i vicini con sconto crescente
    for nbr in self.G.neighbors(last):
        if nbr not in chain:
            d = self.G[last][nbr]['discount']
            if d > prev_disc:
                chain.append(nbr)
                self._ricorsione(chain)
                chain.pop()
```

2F.3 Funzione di supporto (getScore)

```
def getScore(self, chain):
    """
    Ritorna lo sconto medio degli archi del percorso.
    """
    discounts = [
        self.G[chain[i]][chain[i+1]]['discount']
        for i in range(len(chain)-1)
    ]
    return sum(discounts) / len(discounts) if discounts else 0.0
```