Appunti Esame:

- **SQL**

Esempio di query per recuperare gli archi di un grafo tramite join delle stesse tabelle più volte, così da poter ottenere delle coppie. In questo caso nessuna delle tabelle fornite esprimeva la relazione tra due album.

```
SELECT t1.album_id as a1, t2.album_id as a2
FROM track t1, track t2, playlist_track pt1, playlist_track pt2
WHERE t1.album_id != t2.album_id and t1.id = pt1.track_id and
        t2.id = pt2.track_id and pt1.playlist_id = pt2.playlist_id
GROUP BY t1.album_id, t2.album_id
HAVING count(*) >0
```

Esempio di query semplice (parametrizzata) per mappare un'intera tabella con una classe, semplice vincolo imposto su un attributo.

```
SELECT *
FROM team
WHERE year = %s
```

Esempio di query per ottenere una buona struttura degli archi di un grafo, talvolta gli archi stessi, mediante l'uso di *LEAST* e *GREATEST*, che date le due stesse colonne come argomenti, riordinano i dati nell'ordine desiderato. Queste due parole chiave possono essere utili in due casi:

- Se so che nella mia tabella ci sono righe potenzialmente duplicate, che invertono magari solo punto di partenza e punto di arrivo, ma mantengono tutti gli altri attributi uguali, e so a priori di dover considerare quel tale percorso una volta sola.

id_origine	id_destinazione	peso
1	2	5

2	1	5

```
SELECT
    LEAST(id_hub_origine, id_hub_destinazione)    AS h1,
    GREATEST(id_hub_origine, id_hub_destinazione) AS h2,
    MAX(peso) AS peso #oppure AVG tanto sono uguali per costruzione
FROM tratta
GROUP BY h1, h2;
```

- Se so che nella mia tabella ci sono righe che invertono punti di partenza e di arrivo ma sono tutte importanti per me, perché magari devo fare un calcolo aggregato su tutto ciò che avviene lungo una certa tratta, indifferentemente dalla direzione.

id_origine	id_destinazione	peso
1	2	5
2	1	7

```
SELECT
    LEAST(id_hub_origine, id_hub_destinazione)    AS h1,
    GREATEST(id_hub_origine, id_hub_destinazione) AS h2,
    SUM(peso) AS peso_totale #oppure AVG a seconda della richiesta
FROM tratta
GROUP BY h1, h2
```

Esempio di query annidata, per recuperare le informazioni sui nodi.

*“I vertici devono rappresentare gli aeroporti su cui operano almeno **x** compagnie aeree (in arrivo o in partenza)”.*

Nella subquery dunque vengono contati i voli per ogni aeroporto e per ogni compagnia aerea, nella query esterna vengono contate le compagnie aeree per ogni aeroporto.

```
SELECT tmp.id, tmp.IATA_CODE, count(*) as somma
FROM
```

```
        (SELECT a.id, a.IATA_CODE, f.AIRLINE_ID, COUNT(*)
    FROM flights f, airports a
    WHERE a.id = f.ORIGIN_AIRPORT_ID OR
        a.id = f.DESTINATION_AIRPORT_ID
    GROUP BY a.id, a.IATA_CODE, f.AIRLINE_ID) AS tmp

GROUP BY tmp.id, tmp.IATA_CODE
HAVING somma>= %s
```

Molto spesso, specialmente quando vengono mappate le tabelle con delle classi, conviene creare dei dizionari che abbiano come chiavi l'attributo indicato come hashable della tabella, che sia un id o un nome, e far corrispondere a tali chiavi gli oggetti veri e propri, così da facilitare query successive e l'accesso ai dati nell'elaborazione in python.

- **Flet**

Bottoni.

```
View:

self.pulsante_percorso = ft.ElevatedButton(text="Percorso",
 on_click=self.controller.handle_percorso)

Controller: #importantissimo l'uso dell' "e" tra gli arguments della funzione

 def handle_percorso(self, e):
        self._model.cerca_percorso()
```

ListView.

```
self._view.txt_out_squadre.controls.append(ft.Text(f"Numero squadre: \
                                                    {len(self._model.lista_squ
adre)}"))

#Due modalità per la pulizia della ListView:
```

```
self._view.txt_out_squadre.controls.clear()
self._view.txt_risultato.clean()
```

DropDown.

Qualora fosse necessario dover mostrare dei valori nella view popolando una DD all'avvio, assicurarsi che la funzione per popolarla venga chiamata dentro la view immediatamente dopo che la DD stessa è stata inizializzata. Siccome la view nel main è l'ultima ad essere generata, si andrà poi a ritroso nell'esecuzione dei comandi nel controller e poi nel model.

```
#Popolare la DD:

self._view.dd_squadra.options.append(ft.dropdown.Option(text=f"{team.tea
m_code}\
                                                ({team.name})",key=team.id))

"""Quindi quando seleziono un elemento nella dropdown io ne vedo il nom
e,
ma quando vado a leggere il value dopo aver intercettato l'evento "on chan
ge"
io leggo la key che è stata inserita (SEMPRE COME STRINGA)."""

#Creazione della DD (View):

self.dd_squadra = ft.Dropdown(label="Squadra", width=200,
                                          on_change=self.controller.set_squadra)

#Funzione event handler (e) della DD (Controller):

def set_squadra(self, e):
    self.squadra = e.control.value

#Pulizia della DD:

self._view.dd_squadra.options.clear()
```

TextField.

```
View:

self.txt_anno = ft.TextField(label="Anno (1950-2024)", width=200)

Controller:

def handle_grafo(self, e):
     """Callback per il pulsante 'Crea Grafo'."""
   try:
      anno = int(self._view.txt_anno.value)
   except:
      self._view.show_alert("Inserisci un numero valido per l'anno.")
      return

   if anno < 1950 or anno > 2024:
      self._view.show_alert("Anno fuori intervallo (1950-2024).")
      return
```

Nell’impostazione dei valori selezionati dall’utente, specie sull’on_click di bottoni e alla lettura dei value nei TextField, impostare dei try/except per la verifica della correttezza dei dati immessi come sopra.

- **NetworkX**

Buona prassi prima della creazione di un qualsiasi tipo di grafo è pulire il grafo e le strutture dati collegate:

```
def build_graph(self):
      self.G.clear()
```

Analisi delle tipologie di grafo:

- Grafo semplice, non orientato, eventualmente pesato:
    - Tra due nodi può esistere un solo arco, non essendoci direzioni non fa differenza considerare l’arco (1,2) piuttosto che l’arco (2,1) perché sono

la stessa cosa.

```
self.G = nx.Graph()
```

- Grafo semplice, orientato, eventualmente pesato:
  - Tra due nodi posso avere massimo due archi, uno per ciascuna direzione, gli archi (1,2) e (2,1) sono a tutti gli effetti diversi .

```
self.G = nx.DiGraph()
```

- Multigrafo, non orientato, eventualmente pesato:
  - Tra due nodi posso avere più di un arco, l'unico modo per distinguerli l'uno dall'altro è specificare una key per l'arco, non conta la direzione.

```
self.G = nx.MultiGraph()

self.G.add_edge(1, 2, key="playlist_42", weight=5)
self.G.add_edge(1, 2, key="playlist_17", weight=8)

# → (1,2, key="a") e (2,1, key="a") → stesso arco
# → (1,2, key="a") e (1,2, key="b") → archi distinti
```

- Multigrafo, orientato, eventualmente pesato:
  - Tra due nodi posso avere al più due archi con la stessa chiave, ma in generale archi multipli che li collegano. Una stessa chiave può essere riferita solo a due archi i quali vanno l'uno in direzione opposta all'altro.
  - Per ogni coppia ordinata di nodi (u, v) possono esistere più archi distinti, identificati tramite una key. Un arco è univocamente identificato dalla terna (u, v, key). La stessa key può essere riutilizzata per archi tra gli stessi nodi ma con direzione opposta.

```
self.G = nx.MultiDiGraph()
```

Creazione del grafo e gestione dei suoi elementi:

- Aggiunta di nodi e archi, possibili soluzioni:

1. Aggiunta dei nodi:

   self.g.add_nodes_from(self.lista_squadre)

2. Aggiunta degli archi:

   self.DiGraph.add_edge(u_of_edge, v_of_edge, **attr) attributo opzionale

```
G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
e = (1, 2)
G.add_edge(1, 2)  # explicit two-node form
G.add_edge(*e)  # single edge as tuple of two nodes
G.add_edges_from([(1, 2)])  # add edges from iterable container

G.add_edge(1, 2, weight=3)
G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

- Accesso agli elementi del grafo:

1. Se voglio accedere ad un particolare peso di un arco:

   valore_peso = self.g[u][v]['weight']

   Accesso iterando:

```
for u, v, data in G.edges(data=True):
    print(u, v, data["weight"])
```

2. Accesso ai vicini di un nodo:

```
    list(G.neighbors(nodo))

Oppure:

for n in G.neighbors(nodo):
    print(n)
```

3. DiGraph - Archi entranti e uscenti:

```
for n in self.G.nodes:
    score = 0
    for e_out in self.G.out_edges(n, data=True):
        score += e_out[2]["weight"]
    for e_in in self.G.in_edges(n, data=True):
        score -= e_in[2]["weight"]
```

4. DiGraph - Nodi adiacenti, accesso con comandi di libreria e archi:

I comandi che seguono mi restituiscono rispettivamente l'insieme dei n
odi
da cui partono archi che sono diretti verso "nodo" (pred) e l'insieme dei
nodi che sono raggiunti da archi che partono da "nodo" (succ).

```
insieme_pred = set(G.predecessors(nodo))
insieme_succ = set(G.successors(nodo))

oppure:

set(G.in_edges(nodo)) | set(G.out_edges(nodo))
```

5. MultiGraph o MultiDiGraph:

```
for u, v, k, data in G.edges(keys=True, data=True):
    print(u, v, k, data["weight"])
```

```
peso = G[u][v][key]["weight"]

Dizionario degli attributi di un arco:

diz_attributi = self.g[u][v]  (es. {'weight':5, 'color':'red'})
```

- Visualizzazione delle informazioni sul grafo:

```
list(G.nodes())
list(G.edges())

Con attributi:

G.nodes(data=True)
G.edges(data=True)

MultiGraph:

G.edges(keys=True, data=True)


Ordinamento dei vicini di un nodo, per peso decrescente degli archi:

def get_neighbors(self, team):
    vicini = []
    for n in self.G.neighbors(team):
        w = self.G[team][n]["weight"]
        vicini.append((n, w))
    return sorted(vicini, key=lambda x: x[1], reverse=True)
```

- Costruzione di algoritmi ricorsivi

  Scheletro del codice + comando di libreria NetworkX per la verifica del risultato

  - Ricerca del cammino minimo (con meno archi) tra due nodi:

```
import copy

def cammino_minimo_ricorsivo(G, start, end):
    miglior_cammino = []  # risultato finale
    visitati = set()

    def ricorsione(parziale):
        nonlocal miglior_cammino

        nodo_corrente = parziale[-1]

        # Caso base: siamo arrivati a destinazione
        if nodo_corrente == end:
            # Se non esiste cammino migliore o il nuovo è più corto
            if not miglior_cammino or len(parziale) < len(miglior_cammino):
                miglior_cammino = copy.deepcopy(parziale)
            return

        for vicino in G.neighbors(nodo_corrente):
            if vicino not in parziale:  # evita cicli
                parziale.append(vicino)
                ricorsione(parziale)
                parziale.pop()  # backtracking

    ricorsione([start])
    return miglior_cammino


# --------- Esempio d'uso ----------
# G è un networkx.Graph non orientato
# start e end sono nodi di G
cammino = cammino_minimo_ricorsivo(G, 'A', 'F')
print("Cammino minimo:", cammino)


# G non pesato o consideri tutti i pesi = 1
```

```
cammino = nx.shortest_path(G, source='A', target='F')
print("Cammino minimo con NetworkX:", cammino)
```

- Ricerca del cammino di peso minimo tra due nodi:

```
import copy

def cammino_peso_minimo_ricorsivo(G, start, end):
    """
    Restituisce il cammino con peso totale minimo tra start ed end
    """
    miglior_cammino = [None]  # conterrà la lista dei nodi del cammino \
                                        # migliore

    peso_minimo = [float('inf')]  # peso minimo trovato finora

    def ricorsione(parziale, peso_corrente):
        nodo_corrente = parziale[-1]

        # Caso base: siamo arrivati a destinazione
        if nodo_corrente == end:
            if peso_corrente < peso_minimo[0]:
                miglior_cammino[0] = copy.deepcopy(parziale)
                peso_minimo[0] = peso_corrente
            return

        # Esplora tutti i vicini
        for vicino in G.neighbors(nodo_corrente):
            if vicino not in parziale:  # evita cicli
                arco_peso = G[nodo_corrente][vicino].get('weight', 1)
                parziale.append(vicino)
                ricorsione(parziale, peso_corrente + arco_peso)
                parziale.pop()  # backtracking

    # Avvia ricorsione
    ricorsione([start], 0)
    return miglior_cammino[0], peso_minimo[0]
```

```
# --------- Esempio d'uso ----------
# G = grafo pesato NetworkX
start = 'A'
end = 'F'
cammino, peso = cammino_peso_minimo_ricorsivo(G, start, end)
print(f"Cammino di peso minimo tra {start} e {end}: {cammino} \
        (peso totale: {peso})")



cammino = nx.dijkstra_path(G, source='A', target='F', weight='weight')
print("Cammino minimo pesato:", cammino)
```

- Ricerca del cammino di peso minimo tra tutti i nodi del grafo:

```
import copy

def cammino_peso_minimo_tutti_nodi(G):
    """
        Restituisce un dizionario con il cammino di peso minimo tra tutte
        le coppie di nodi
    """
    cammini_minimi = {}  # dizionario: (start, end) → (cammino, peso)

    nodi = list(G.nodes())

    def ricorsione(parziale, peso_corrente, end, miglior_cammino,\
                                peso_migliore):
        nodo_corrente = parziale[-1]

        if nodo_corrente == end:
            if peso_corrente < peso_migliore[0]:
                miglior_cammino[0] = copy.deepcopy(parziale)
                peso_migliore[0] = peso_corrente
            return

        for vicino in G.neighbors(nodo_corrente):
```

```
            if vicino not in parziale:  # evita cicli
                arco_peso = G[nodo_corrente][vicino].get('weight', 1)
                parziale.append(vicino)
                ricorsione(parziale, peso_corrente + arco_peso, end, \
                                    miglior_cammino, peso_migliore)
                parziale.pop()

    # ciclo su tutte le coppie (start, end)
    for i in range(len(nodi)):
        for j in range(i+1, len(nodi)):
            start = nodi[i]
            end = nodi[j]
            miglior_cammino = [None]
            peso_migliore = [float('inf')]
            ricorsione([start], 0, end, miglior_cammino, peso_migliore)
            if miglior_cammino[0]:
                cammini_minimi[(start, end)] = (miglior_cammino[0],\
                peso_migliore[0])

    return cammini_minimi


# --------- Esempio d'uso ----------
# G = grafo pesato NetworkX
cammini = cammino_peso_minimo_tutti_nodi(G)
for coppia, (percorso, peso) in cammini.items():
    print(f"{coppia}: cammino={percorso}, peso={peso}")



# Cammini minimi pesati tra tutte le coppie
cammini_pesati = dict(nx.all_pairs_dijkstra_path(G, weight='weight'))

# Lunghezze dei cammini
lunghezze_pesate = dict(nx.all_pairs_dijkstra_path_length(G, weight='w
eight'))

# Stampa
```

```
for start, targets in cammini_pesati.items():
    for end, percorso in targets.items():
        peso = lunghezze_pesate[start][end]
        print(f"{start} → {end}: cammino={percorso}, peso={peso}")
```

- Ricerca del cammino di peso massimo tra due nodi, noti entrambi:

In questo esercizio si ha il vincolo che il percorso abbia esattamente una certa lunghezza "lungh", se non ci fosse basterebbe rimuovere il parametro e la condizione nella ricorsione.

```
def get_best_path(self, lungh, start, end):
        self.best_path = []
        self.best_score = 0
        parziale = [start]
        self._ricorsione(parziale, lungh, start, end)
        return self.best_path, self.best_score

    def _ricorsione(self, parziale, lungh, start, end):
        if len(parziale) == lungh:
            if  parziale[-1] == end and self._get_score(parziale) > self.best_s
core:
                self.best_score = self._get_score(parziale)
                self.best_path = copy.deepcopy(parziale)
            return

        for n in self.G.successors(parziale[-1]):
            if n not in parziale:
                parziale.append(n)
                self._ricorsione(parziale, lungh, start, end)
                parziale.pop()

    def _get_score(self, parziale):
        score = 0
        for i in range(1, len(parziale)):
```

```
            score += self.G[parziale[i-1]][parziale[i]]["weight"]
        return score
```

- Ricerca del cammino di peso massimo tra due nodi del grafo con numero massimo di archi visitabili, è noto solo il nodo di partenza:

```
def get_neighbors(self, team):
        vicini = []
        for n in self.G.neighbors(team):
            w = self.G[team][n]["weight"]
            vicini.append((n, w))
        return sorted(vicini, key=lambda x: x[1], reverse=True)

def compute_best_path(self, start):
    """Calcola percorso di peso massimo con archi strettamente decresc
enti"""
        self.best_path = []
        self.best_weight = 0
        self._ricorsione([start], 0, float("inf"))
        return self.best_path, self.best_weight

    def _ricorsione(self, path, weight, last_edge_weight):
        last = path[-1]
        if weight > self.best_weight:
            self.best_weight = weight
            self.best_path = path.copy()

        vicini = self.get_neighbors(last)
        neighbors = []
        counter = 0
        for node, edge_w in vicini:
            if node in path:
                continue
            if edge_w <= last_edge_weight:
                neighbors.append((node, edge_w))
                counter += 1
                if counter == self.K:
                    break
```

```
        for node, edge_w in neighbors:
            path.append(node)
            self._ricorsione(path, weight + edge_w, edge_w)
            path.pop()
```

- Ricerca del cammino di peso massimo tra tutti i nodi del grafo:

```
import copy

def cerca_cammino_peso_massimo(self):
    self.cammino_ottimo = []
    self.peso_ottimo = 0.0

    for nodo_start in self.G.nodes():
        parziale = [nodo_start]
        self._ricorsione(parziale)

    return self.cammino_ottimo, self.peso_ottimo

def _ricorsione(self, parziale):
    peso_corrente = self._get_score(parziale)

    # aggiorna soluzione ottima (cammino con almeno 1 arco)
    if len(parziale) > 1 and peso_corrente > self.peso_ottimo:
        self.peso_ottimo = peso_corrente
        self.cammino_ottimo = copy.deepcopy(parziale)

    # espansione su grafo NON orientato
    for vicino in self.G.neighbors(parziale[-1]):
        if vicino not in parziale:   # evita cicli
            parziale.append(vicino)
            self._ricorsione(parziale)
            parziale.pop()

def _get_score(self, parziale):
    score = 0.0
    for i in range(1, len(parziale)):
```

```
        score += self.G[parziale[i-1]][parziale[i]]["weight"]
    return score
```