lezione09

October 23, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 9 - 23 ottobre 2023

1.0.1 RECAP:

- annotazioni di tipo e type checkers:
  - **mypy**: statico (usabile anche in Spyder e Jupyter)
  - **typeguard**: run-time
  - **pyre-check** (Facebook)
  - **pytype** (Google)
  - **pyright** (Microsoft)
- dizionari ed istogramma

[1]:
```
%load_ext nb_mypy
```

```
Version 1.0.5
```

2 Analisi: Anagrammi

Due parole sono anagrammi se **contengono le stesse lettere lo stesso numero di volte** - Soluzione 1: contiamo le lettere

[2]:
```
def isAnagramma(testo : str, testo1 : str) -> bool:
    # conto i caratteri nelle due stringhe    O(n)
    # confronto i due dizionari               O(n)
    def conta_lettere(stringa : str) -> dict[str,int]:
        frequenze : dict[str,int] = {}
        for c in stringa:
            frequenze[c] = frequenze.get(c, 0) + 1
        return frequenze
    frequenze1 = conta_lettere(testo)
    frequenze2 = conta_lettere(testo1)
    return frequenze1 == frequenze2

isAnagramma('onepaper', 'paperino')
```

[2]: False

2.1 Forma canonica

Altre definizioni, sono anagrammi se: - **una si ottiene dall'altra per spostamento di lettere** - oppure **hanno la stessa forma "standardizzata"** (o "forma canonica")

NOTA: la forma "canonica" deve essere **unica**

Altro esempio di riduzione a forma canonica: **formula booleana ==> forma SOP ==> tabella di verità**

Come **"standardizzare"** le parole in modo che siano uguali? - **ordiniamo le lettere**

[ ]:
```python
def isAnagramma2(testo : str, testo1 : str) -> bool:
    # ordino le due stringhe      O(n log(n))
    # le confronto                O(n)
    ordinata1 = sorted(testo)
    ordinata2 = sorted(testo1)
    return ordinata1 == ordinata2

isAnagramma2('paperone','eaepnrpo')

# Database come liste di dizionari

ciascun dizionario rappresenta una riga
```

3 Analisi: Istogramma delle frequenze

Dato un elenco di dati vogliamo calcolarne le frequenze (quante volte appare ciascun valore) e produrre una stampa in cui visualizziamo le frequenze su righe successive, come **barre di asterischi**

Esempio: da [1, 4, 2, 4, 1, 4, 4, 2, 4, 4, 1, 1, 4] otteniamo

```
1  ****
2  **
3
4  *******
```

3.0.1 Problema: stampare l'istogramma di una serie di dati

Input: - la lista di dati numerici (interi/float?) - la definizione degli intervalli (implicita/esplicita?)

Output: - stringa formata da più righe, una per ogni valore: - ciascuna riga contiene il **valore, spazio** e tanti **asterischi** quante volte quel valore appare nei dati

Da decidere: i valori con conteggio 0 devono apparire?

3.0.2 Istogramma con estremi e dati interi

[2]:
```
'''
# per ottenere l'istogramma conoscendo gli estremi A,B e i dati
    # calcolo le frequenze dei dati
        # opzione uno: usiamo una lista se:
            # i valori sono interi
            # conosciamo l'intervallo di valori possibili [A, B]
            # ciascun 'bin' è largo 1
    # dalle frequenze produco la stringa
'''
def istogramma_con_estremi_e_dati_interi(A : int,
                                         B : int,
                                         dati : list[int]) -> str:
    frequenze : list[int] = calcola_frequenze(A, B, dati)
    return produci_istogramma(A, frequenze)
```

```
<cell>13: error: Name "calcola_frequenze" is not defined
[name-defined]
<cell>14: error: Name "produci_istogramma" is not defined
[name-defined]
```

[3]:
```
# per calcolare le frequenze come lista di conteggi
def calcola_frequenze(A : int, B : int, dati : list[int]) -> list[int] :
    'attenzione agli indici della lista per dati positivi e negativi!'
    # usiamo un offset
    conteggi : list[int] = [0] *(B-A+1)      # devo contenere tutti i valori da␣
 ↪A a B COMPRESI
    for valore in dati:
        if A <= valore <= B: # controllo che il valore sia in [A,B]
            conteggi[valore-A] += 1          # sottraggo A per allinearlo␣
 ↪all'indice 0
    return conteggi
```

[4]:
```
def produci_istogramma(A : int, frequenze : list[int]) -> str :
    testo = ''
    for indice, frequenza in enumerate(frequenze):
        valore     = indice + A          # devo riaggiungere A all'indice
        asterischi = "*"*frequenza
        testo += f"{valore:>4}\t{asterischi}\n"
    return testo

L : list[int] = [ 2, 17, 1, 4, 2, 8, 11, 2, 4, 8, 11, 3 ]
print(istogramma_con_estremi_e_dati_interi(-5, 20, L))
```

```
  -5
  -4
  -3
```

```
-2
-1
 0
 1    *
 2    ***
 3    *
 4    **
 5
 6
 7
 8    **
 9
10
11    **
12
13
14
15
16
17    *
18
19
20
```

3.0.3 Istogramma con estremi e dati float

- in questo caso posso troncare i valori
- oppure li potrei arrotondare all'intero più vicino con **round**

[6]:
```
## WHAT IF i dati sono float?
def istogramma_con_estremi_e_dati_float(A : int, B : int,
                                        dati : list[float]) -> str:
    frequenze = calcola_frequenze_float(A,B, dati)
    return produci_istogramma(A, frequenze)

def calcola_frequenze_float(A : int, B : int,
                            dati : list[float]) -> list[int]:
    dati_interi = [ int(X) for X in dati ]          # tronco i valori oppure␣
 ↪uso round
    return calcola_frequenze(A,B, dati_interi)

LF = [ 1.3, 5.7, 2.1, 5.8, 2.4, 4.2, 1 ]
print(istogramma_con_estremi_e_dati_float(-2,9, LF))
```

```
-2
-1
 0
```

```
1    **
2    **
3
4    *
5    **
6
7
8
9
```

3.0.4 Istogramma usando dizionari

Se i dati sono sparsi e ci sono molte frequenze nulle posso usare meno memoria

[7]:
```python
# se vogliamo usare meno memoria possiamo usare un dizionario
# O(n)
def calcola_frequenze_con_diz_int(dati : list[int]) -> dict[int, int] :
    frequenze : dict[int,int] = {}
    for valore in dati:            # N volte
        frequenze[valore] = frequenze.get(valore, 0) + 1
    return frequenze

# Esempio
from random import choices
L = choices(range(-10,10), k=30)
istogramma = calcola_frequenze_con_diz_int(L)
print(L)
print(istogramma)
```

```
[-1, 3, -3, 1, -5, 5, -1, -2, 0, 4, -2, 7, -10, 0, -4, -5, -4, 8, 4, 8, 8, 8, 1,
7, 3, -8, 4, 4, -1, 0]
{-1: 3, 3: 2, -3: 1, 1: 2, -5: 2, 5: 1, -2: 2, 0: 3, 4: 4, 7: 2, -10: 1, -4: 2,
8: 4, -8: 1}
```

[8]:
```python
# O(n^2)
def calcola_frequenze_con_diz_int2(dati : list[int]) -> dict[int, int] :
    frequenze = {}
    for valore in dati:                          # N volte
        frequenze[valore] = dati.count(valore)   # O(N)
    return frequenze
    # oppure con una list-comprehension
    '''return { dati.count(valore) for valore in dati }'''
```

[9]:
```python
# ricordiamoci di ordinare le chiavi
def produci_istogramma_diz_con_buchi(istogramma : dict[int,int]) -> str :
    # ignoro i valori con conteggio 0
    testo = ''
```

```
    for valore, frequenza in sorted(istogramma.items()):
        testo += f"{valore}\t" + ('*'*frequenza) + '\n'
    return testo

# Esempio
print(produci_istogramma_diz_con_buchi(calcola_frequenze_con_diz_int2(L)))
```

```
-10     *
-8      *
-5      **
-4      **
-3      *
-2      **
-1      ***
0       ***
1       **
3       **
4       ****
5       *
7       **
8       ****
```

[10]:
```
# per visualizzare anche i valori con frequenza 0 li posso aggiungere al␣
 ↪dizionario
def produci_istogramma_diz_senza_buchi(istogramma : dict[int,int]) -> str :
    # generare anche i dati con conteggio 0
    # aggiungiamo al dizionario i valori con conteggio 0
    m = min(istogramma)     # minima chiave
    M = max(istogramma)     # massima chiave
    for X in range(m,M+1):
        if not X in istogramma:
            istogramma[X] = 0
    return produci_istogramma_diz_con_buchi(istogramma)

# Esempio
print(produci_istogramma_diz_senza_buchi(calcola_frequenze_con_diz_int2(L)))
```

```
-10     *
-9
-8      *
-7
-6
-5      **
-4      **
-3      *
-2      **
-1      ***
```

```
0       ***
1       **
2
3       **
4       ****
5       *
6
7       **
8       ****
```

[11]:
```python
# oppure ne posso tenere conto nella generazione del testo
# come frequenza = 0 se la chiave non è presente
def produci_istogramma_diz_senza_buchi2(istogramma : dict[int,int]) -> str :
    m = min(istogramma)      # minima chiave
    M = max(istogramma)      # massima chiave
    testo = ''
    for X in range(m,M+1):
        frequenza = istogramma.get(X, 0)      # se X non è nel dizionario torna frequenza 0
        asterischi = '*'*frequenza
        testo += f"{X:>4}\t{asterischi}\n"
    return testo

print(produci_istogramma_diz_senza_buchi2(istogramma))
```

```
-10     *
 -9
 -8     *
 -7
 -6
 -5     **
 -4     **
 -3     *
 -2     **
 -1     ***
  0     ***
  1     **
  2
  3     **
  4     ****
  5     *
  6
  7     **
  8     ****
```

4 Analisi: Realizzazione di un database come lista di dizionari

- ciascuna scheda è un dizionario **info -> valore**
- il database è una lista di schede

[12]:
```
Scheda = dict[str, str]  # una Scheda è un dizionario 'informazione' -> 'valore'
Agenda = list[Scheda]    # una Agenda è una lista di Schede

agenda : Agenda = [
    {'nome'     : 'Paperino',
     'cognome'  :'Paolino',
     'telefono' :'555-1313',
     'indirizzo': 'via dei Peri 113',
     'città'    : 'Paperopoli'},
    {'nome': 'Gastone', 'cognome':'Paperone',     'telefono':'555-1717', ␣
 ↪'indirizzo': 'via dei Baobab 42',         'città': 'Paperopoli'},
    {'nome': 'Paperon', 'cognome':"de' Paperoni", 'telefono':'555-99999',␣
 ↪'indirizzo': 'colle Papero 1',            'città': 'Paperopoli'},
    {'nome': 'Archimede','cognome':'Pitagorico',  'telefono':'555-11235',␣
 ↪'indirizzo': 'colle degli Inventori 1',   'città': 'Paperopoli'},
    {'nome': 'Pietro',  'cognome':'Gambadilegno', 'telefono':'555-66666',␣
 ↪'indirizzo': 'via dei Ladri 13',          'città': 'Topolinia'},
    {'nome': 'Trudy',   'cognome':'Gambadilegno', 'telefono':'555-66666',␣
 ↪'indirizzo': 'via dei Ladri 13',          'città': 'Topolinia'},
    {'nome': 'Topolino','cognome':'Mouse',        'telefono':'555-12345',␣
 ↪'indirizzo': 'via degli Investigatori 1', 'città': 'Topolinia'},
    {'nome': 'Minnie',  'cognome':'Mouse',        'telefono':'555-54321',␣
 ↪'indirizzo': 'via di M.me Curie 1',       'città': 'Topolinia'},
    {'nome': 'Pippo',   'cognome':"de' Pippis",   'telefono':'555-33333',␣
 ↪'indirizzo': 'via dei Pioppi 1',          'città': 'Topolinia'},
    ]
```

4.0.1 Ricerca di un record lineare

[15]:
```
from typing import Optional
# per cercare un numero di telefono (un record)
# sapendo quale colonna usare e quale valore cerchiamo
def cerca_lineare(ag : Agenda,
                  colonna : str,
                  valore : str) -> Optional[Scheda] :
    # per tutti i record dell'agenda
    for record in ag:
        # se *la colonna esiste ed inoltre contiene il valore cercato*
        if corrisponde_alla_query(record, colonna, valore):
            # torniamo il record
            return record
    # altrimenti alla fine torniamo None
```

```
    return None

cerca_lineare(agenda, 'cognome', 'Pitagorico')
```

[15]:
```
{'nome': 'Archimede',
 'cognome': 'Pitagorico',
 'telefono': '555-11235',
 'indirizzo': 'colle degli Inventori 1',
 'città': 'Paperopoli'}
```

[14]:
```
# per vedere se un record corrisponde alla query (nome della colonna e valore␣
 ↪cercato)
def corrisponde_alla_query(riga    : Scheda,
                           colonna : str,
                           valore  : str) -> bool:
    return (colonna in riga                # la colonna deve esistere
            and riga[colonna] == valore)   # il valore deve corrispondere
```

4.0.2 Ricerca con query multicolonna

[18]:
```
# una query è un dizionario colonna -> valore
# per cercare su più colonne
def cerca_multicolonna_lineare(
    ag : Agenda, query : Scheda) -> list[Scheda] :
    # IN : agenda, query: coppie colonna/valore (con un dizionario)
    # OUT: lista dei record che soddisfano tutte le condizioni
    # all'inizio l'elenco di record risultante è []
    trovati = []
    for record in ag: # scandiamo l'agenda e per tutti i record
        # se *corrispondono alla query*
        if corrisponde_alla_query_multicolonna(record, query):
            trovati.append(record) # aggiungo il record all'output
    # torno l'elenco di record
    return trovati
```

```
<cell>11: error: Name
"corrisponde_alla_query_multicolonna" is not defined  [name-
defined]
```

[19]:
```
# con una list comprehension
def cerca_multicolonna_lineare(
    ag : Agenda, query : Scheda) -> list[Scheda] :
    return [record  for record in ag
                    if corrisponde_alla_query_multicolonna(record,query) ]
```

```
<cell>5: error: Name "corrisponde_alla_query_multicolonna"
is not defined  [name-defined]
```

[22]:
```python
# per determinare se un record corrisponde ad una query
def corrisponde_alla_query_multicolonna(
    record : Scheda, query : Scheda ) -> bool :
    # per ciascuna delle coppie colonna : valore cercato
    for colonna, cercato in query.items():
        # se colonna NON è nel record e o non ha quel valore
        if not corrisponde_alla_query(record, colonna, cercato):
            # torno False
            return False
    # altrimenti alla fine torno True
    return True

# MA QUESTO NON E' ALTRO CHE UN AND LOGICO!

Q = { 'città' : 'Topolinia',
     'cognome' : 'Gambadilegno'  }
cerca_multicolonna_lineare(agenda,Q)
```

[22]:
```
[{'nome': 'Pietro',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'},
 {'nome': 'Trudy',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'}]
```

4.1 Stile di programmazione funzionale

- **all** : torna True se TUTTI i valori sono True
- **any** : torna True se ALMENO un valore è True
- **filter** : torna solo gli elementi per cui è vero un **predicato**
- **map** : torna una "lista" di elementi trasformati
- …

[26]:
```python
# per determinare se un record corrisponde ad una query
# versione funzionale
def corrisponde_alla_query_FUN(
    record : Scheda, query : Scheda ) -> bool :
    return all(          # tutte le info della query devono essere vere
        corrisponde_alla_query(record, chiave, valore)
        for chiave, valore in query.items()
    )

R = {'1':'a', '2':'b', '3':'c'}
Q = {'2':'b', '1':'a'}
```

```
corrisponde_alla_query_FUN(R, Q)
```

[26]: True

[27]:
```
# per determinare se un record corrisponde ad una query
# versione con insiemi
def corrisponde_alla_query_SET(
    record : Scheda, query : Scheda ) -> bool :
    # L'intersezione di record e query è == query
    # ovvero query - record == set vuoto
    set_query  = set(query.items())
    set_record = set(record.items())
    return set_query - set_record == set()

corrisponde_alla_query_SET(R,Q)
```

[27]: True

[28]:
```
# per cercare su più colonne con list comprehension
def cerca_multicolonna_lineare_LC(
    ag : Agenda, query : Scheda) -> list[Scheda] :
    return [
        record for record in ag
               if corrisponde_alla_query_FUN(record, query)
    ]

Q = { 'città' : 'Topolinia',
     'cognome' : 'Gambadilegno'  }
cerca_multicolonna_lineare_LC(agenda,Q)
```

[28]:
```
[{'nome': 'Pietro',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'},
 {'nome': 'Trudy',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'}]
```

[30]:
```
# usando una funzione filtro che sceglie cosa tenere
def cerca_multicolonna_lineare_filter(ag : Agenda, query : Scheda) ->␣
 ↪list[Scheda] :
    # filtro i record con la funzione c_a_q
    # che usa implicitamente il parametro query
    def c_a_q(record):
```

```
        return corrisponde_alla_query_SET(record, query) # le "inner functions"␣
 ↪possono leggere il namespace locale
    return list(filter(c_a_q, ag))

cerca_multicolonna_lineare_filter(agenda,Q)
```

[30]:
```
[{'nome': 'Pietro',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'},
 {'nome': 'Trudy',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'}]
```

[31]:
```
def cerca_multicolonna_lineare_lambda(ag : Agenda, query : Scheda) ->␣
 ↪list[Scheda] :
    # lo stesso si può fare con una lambda
    return list(filter(
        lambda record: corrisponde_alla_query_SET(record, query),
        ag))
```

4.2 Ordinamento delle schede

[33]:
```
# per ordinare una agenda rispetto ad una colonna
def ordina_rispetto_a_colonna(ag : Agenda, colonna : str) -> Agenda :
    # usiamo sorted con il parametro key
    # che deve tornare il valore rispetto al quale vogliamo ordinare
    def criterio(riga : Scheda) -> str:
        return riga.get(colonna, '')  # se la colonna non esiste sono tutte␣
 ↪equivalenti
    return sorted(ag, key=criterio)

print(*ordina_rispetto_a_colonna(agenda, 'cognome'), sep='\n')
```

```
{'nome': 'Pietro', 'cognome': 'Gambadilegno', 'telefono': '555-66666',
'indirizzo': 'via dei Ladri 13', 'città': 'Topolinia'}
{'nome': 'Trudy', 'cognome': 'Gambadilegno', 'telefono': '555-66666',
'indirizzo': 'via dei Ladri 13', 'città': 'Topolinia'}
{'nome': 'Topolino', 'cognome': 'Mouse', 'telefono': '555-12345', 'indirizzo':
'via degli Investigatori 1', 'città': 'Topolinia'}
{'nome': 'Minnie', 'cognome': 'Mouse', 'telefono': '555-54321', 'indirizzo':
'via di M.me Curie 1', 'città': 'Topolinia'}
{'nome': 'Paperino', 'cognome': 'Paolino', 'telefono': '555-1313', 'indirizzo':
'via dei Peri 113', 'città': 'Paperopoli'}
{'nome': 'Gastone', 'cognome': 'Paperone', 'telefono': '555-1717', 'indirizzo':
```

```
'via dei Baobab 42', 'città': 'Paperopoli'}
{'nome': 'Archimede', 'cognome': 'Pitagorico', 'telefono': '555-11235',
'indirizzo': 'colle degli Inventori 1', 'città': 'Paperopoli'}
{'nome': 'Paperon', 'cognome': "de' Paperoni", 'telefono': '555-99999',
'indirizzo': 'colle Papero 1', 'città': 'Paperopoli'}
{'nome': 'Pippo', 'cognome': "de' Pippis", 'telefono': '555-33333', 'indirizzo':
'via dei Pioppi 1', 'città': 'Topolinia'}
```

[34]:
```python
# per ordinare una agenda rispetto ad una colonna
# stavolta usando lambda
def ordina_rispetto_a_colonna_L(ag : Agenda, colonna : str) -> Agenda :
    # usiamo sorted con il parametro key
    # che deve tornare il valore rispetto al quale vogliamo ordinare
    return sorted(ag, key=lambda riga: riga.get(colonna, ''))

print(*ordina_rispetto_a_colonna_L(agenda, 'cognome'), sep='\n')
```

```
{'nome': 'Pietro', 'cognome': 'Gambadilegno', 'telefono': '555-66666',
'indirizzo': 'via dei Ladri 13', 'città': 'Topolinia'}
{'nome': 'Trudy', 'cognome': 'Gambadilegno', 'telefono': '555-66666',
'indirizzo': 'via dei Ladri 13', 'città': 'Topolinia'}
{'nome': 'Topolino', 'cognome': 'Mouse', 'telefono': '555-12345', 'indirizzo':
'via degli Investigatori 1', 'città': 'Topolinia'}
{'nome': 'Minnie', 'cognome': 'Mouse', 'telefono': '555-54321', 'indirizzo':
'via di M.me Curie 1', 'città': 'Topolinia'}
{'nome': 'Paperino', 'cognome': 'Paolino', 'telefono': '555-1313', 'indirizzo':
'via dei Peri 113', 'città': 'Paperopoli'}
{'nome': 'Gastone', 'cognome': 'Paperone', 'telefono': '555-1717', 'indirizzo':
'via dei Baobab 42', 'città': 'Paperopoli'}
{'nome': 'Archimede', 'cognome': 'Pitagorico', 'telefono': '555-11235',
'indirizzo': 'colle degli Inventori 1', 'città': 'Paperopoli'}
{'nome': 'Paperon', 'cognome': "de' Paperoni", 'telefono': '555-99999',
'indirizzo': 'colle Papero 1', 'città': 'Paperopoli'}
{'nome': 'Pippo', 'cognome': "de' Pippis", 'telefono': '555-33333', 'indirizzo':
'via dei Pioppi 1', 'città': 'Topolinia'}
```

4.3 Tempi

tutti i tempi sono lineari, si può far meglio?

4.3.1 potremmo costruire un indice ...

- usando un dizionario **valore -> lista di indici**
- con velocità di ricerca molto alta (O(1) nel caso medio)

[36]:
```python
# un indice è un dizionario valore -> lista di posizioni nella Agenda che lo␣
 ↪contengono
Indice = dict[str,list[int]]
# per costruire un indice
```

```python
def crea_indice(ag: Agenda, colonna : str ) -> Indice:
    # IN agenda, colonna
    # OUT lista che contiene coppie [valori]:
    #     posizioni originali dei record che contengono quel valore
    # all'inizio il dizionario valori:posizioni è vuoto
    indice : Indice = {}
    # per ogni record dell'agenda e sua posizione,
    for i,record in enumerate(ag):
        # estraggo dal record il valore per quella colonna
        valore = record.get(colonna, '')
        # e aggiungo la posizione alla lista associata (vuota se non c'è)
        indice[valore] = indice.get(valore, []) + [i]
    # ritorno il dizionario valori: posizioni
    return indice

indice_cognome = crea_indice(agenda, 'città')
print(indice_cognome)
```

```
{'Paperopoli': [0, 1, 2, 3], 'Topolinia': [4, 5, 6, 7, 8]}
```

[37]:
```python
## Visto che ci siamo costruiamo TUTTI gli indici
Indici = dict[str , Indice]
colonne = ['nome','cognome','telefono','indirizzo','città']
indici : Indici = {
    colonna : crea_indice(agenda, colonna)
        for colonna in colonne
}
indici
```

[37]:
```
{'nome': {'Paperino': [0],
  'Gastone': [1],
  'Paperon': [2],
  'Archimede': [3],
  'Pietro': [4],
  'Trudy': [5],
  'Topolino': [6],
  'Minnie': [7],
  'Pippo': [8]},
 'cognome': {'Paolino': [0],
  'Paperone': [1],
  "de' Paperoni": [2],
  'Pitagorico': [3],
  'Gambadilegno': [4, 5],
  'Mouse': [6, 7],
  "de' Pippis": [8]},
 'telefono': {'555-1313': [0],
  '555-1717': [1],
  '555-99999': [2],
```

```
  '555-11235': [3],
  '555-66666': [4, 5],
  '555-12345': [6],
  '555-54321': [7],
  '555-33333': [8]},
 'indirizzo': {'via dei Peri 113': [0],
  'via dei Baobab 42': [1],
  'colle Papero 1': [2],
  'colle degli Inventori 1': [3],
  'via dei Ladri 13': [4, 5],
  'via degli Investigatori 1': [6],
  'via di M.me Curie 1': [7],
  'via dei Pioppi 1': [8]},
 'città': {'Paperopoli': [0, 1, 2, 3], 'Topolinia': [4, 5, 6, 7, 8]}}
```

4.4 Ricerca tramite indici

- ricerca singola
- ricerca multipla

[39]:
```
## Per cercare con ricerca singola avendo l'indice
def cerca_con_indice( ag : Agenda, index : Indice, valore : str ):
    # se il valore è nell'indice
    # torno tutti i record nelle posizioni indicate
    # se non è nell'indice torno lista vuota
    return [ ag[i] for i in index.get(valore,[]) ]

cerca_con_indice(agenda, indici['cognome'], 'Gambadilegno')
```

[39]:
```
[{'nome': 'Pietro',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'},
 {'nome': 'Trudy',
  'cognome': 'Gambadilegno',
  'telefono': '555-66666',
  'indirizzo': 'via dei Ladri 13',
  'città': 'Topolinia'}]
```

[41]:
```
# Nella ricerca multi-colonna mi servono tutte le Schede
# che appaiono in tutti gli indici che contengono i valori cercati
def cerca_con_indici( ag : Agenda, indexes : Indici, query: Scheda ) -> Agenda :
    # inizialmente sono valide tutte le righe
    posizioni_ok = set(range(len(ag)))   # tutte le posizioni possibili
    # per ciascuna coppia   colonna -> valore    della query
    for colonna,valore in query.items():
        # se il nome della colonna non appare negli indici torno []
```

```
        if colonna not in indexes: return []
        # se il valore non appare nell'indice della colonna torno []
        if valore not in indexes[colonna]: return []
        # altrimenti prendo gli insiemi delle posizioni
        # corrispondenti ai valori delle colonne
        righe_ok = indexes[colonna][valore]
        posizioni_ok &= set(righe_ok)        #  e li interseco
    # alla fine torno la lista di righe rimaste nell'insieme
    # delle posizioni che ora soddisfano tutte le chiave->valore
    return [ ag[i] for i in posizioni_ok ]

cerca_con_indici(agenda, indici, {'cognome':'Mouse',
                                  'città':'Topolinia'})
```

[41]:
```
[{'nome': 'Topolino',
  'cognome': 'Mouse',
  'telefono': '555-12345',
  'indirizzo': 'via degli Investigatori 1',
  'città': 'Topolinia'},
 {'nome': 'Minnie',
  'cognome': 'Mouse',
  'telefono': '555-54321',
  'indirizzo': 'via di M.me Curie 1',
  'città': 'Topolinia'}]
```