Esercizio 1

Aggiungere alla classe MultiInsieme dell'esercizio visto alla lezione del 5 dicembre scorso, disponibile nei file
2024_12_05 lab A-L es1.py
2024_12_05 lab M-Z es1.py
i metodi

- **unione(self, b:MultiInsieme) -> MultiInsieme**
- **intersezione(self, b:MultiInsieme) -> MultiInsieme**
- **differenza(self, b:MultiInsieme) -> MultiInsieme**

che restituiscono un nuovo multiinsieme eseguendo l'operazione indicata (definite come indicato nell'es. P8.13 a pag, 541 del libro di testo).

Esercizio 2

Un numero razionale è un numero che può essere espresso come frazione in cui numeratore e denominatore sono due numeri interi. Ad esempio 3/4 e -7/100 sono numeri razionali.
Si progetti una classe **Razionale** con due variabili d'istanza intere per rappresentare il numeratore ed il denominatore del numero razionale. (Nota: non è necessario che i numeri razionali siano ridotti, ad esempio 1/2 e 2/4 sono ugualmente accettabili).

Si progettino un costruttore ed i seguenti metodi della classe Razionale:

- proprietà in lettura per **numeratore** e **denominatore**
- proprietà **ridotta** che restituisce una nuova frazione equivalente a quella su cui la proprietà è invocata, che sia però ridotta ai minimi termini (si sfrutti il metodo math.gcd(a,b) della liberia math)
- metodo **__eq__ (self, b: object) -> bool**: restituisce True se o è un intero o un Razionale e il numero razionale self è equivalente al razionale (o l'intero) b, False altrimenti;
- metodo **__gt__ (self, b: Razionale|int) -> bool**: restituisce True se il numero razionale self è maggiore del razionale (o dell'intero) b, False altrimenti;
- in modo simile a __gt__, si implementino **__ge__, __lt__, __le__**
- metodo **__add__(self, b:Razionale|int)-> Razionale**: restituisce un nuovo numero razionale ottenuto sommando a self il Razionale (o l'intero) b.
- in modo simile a __add__, si implementino **__sub__, __mul__, __truediv__**
- metodo **__str__** che rappresenta un razionale come `(a/b)` dove a e b sono il numeratore e il denominatore, rispettivamente
- metodo **__hash__**

Esercizio 3

Si progetti una classe **Statino** con

- una variabili di istanza _**matricola** (tipo int, final, privata)
- una variabile di istanza _**voto** (tipo int, privata).
- una variabile di istanza _**lode** (tipo bool, privata).
- una variabile di istanza _**domande_esame** (tipo String, privata).

La classe deve avere:

- un costruttore che crea uno Statino dati **matricola** e **voto** (se voto è minore di 0 viene impostato a 0; se è maggiore di 30 viene impostato a 30 e viene assegnata la lode) e, opzionalmente, domande_esame (che di default è la stringa vuota);
- proprietà in lettura per voto, matricola, lode
- prorietà in lettura e scrittura per domande_esame il campo _**domande_esame** al valore dato come parametro
- metodi __str__ e __repr__ che restituisce la descrizione nel seguente modo (nell'esempio seguente il campo matricola è 31000000, il campo voto è 24 e il campo domandeEsame è "comando For"):

```
studente 310100000 voto 24 domande: comando For
```

Si progetti una classe **Appello** con

- una variabile di istanza **statini** di tipo set[Statino], private

La classe deve avere:

- un costruttore che crea un appello con nessuno statino (creando l'insieme statini vuoto)
- un metodo **add_statino (self, s: Statino)** che aggiunge all'insieme statino lo statino s
- un metodo **conta_lodi(self) -> int** che restituisce il numero di lodi presenti negli statini.
- un metodo **estrai_matricole_sufficienti(self) -> set[int]** che crea e restituisce un insieme di int contenente tutte le matricole di studenti che hanno riportato un voto sufficiente all'appello.