Corso di Programmazione e Algoritmi 1
Modulo di Laboratorio

A.A. 2018/2019

Progetto

Regole per lo svolgimento:

- Il progetto può essere svolto **singolarmente** o **in gruppi di qualsiasi dimensione**. In ogni caso, ciascuno studente dovrà poi **singolarmente** discutere il progetto dimostrando di essere in grado di modificare e/o aggiungere alcune parti. In caso di esito sufficiente di tale discussione, verrà attribuito il punteggio da 0 a 2 punti.
- Il progetto deve essere consegnato **almeno 7 giorni prima** della prova orale di Laboratorio di Programmazione relativa all'appello in cui si desidera registrare l'esame.
- Il progetto deve contenere le classi, i campi e i metodi richiesti **rispettandone esattamente il nome, il tipo e l'ordine dei parametri formali, ed il tipo di ritorno**.
- Si è naturalmente liberi di sviluppare metodi aggiuntivi di supporto ai metodi richiesti, laddove lo si ritenga utile.
- Deve essere consegnato via email all'indirizzo **luca.moscardelli@unich.it** un file compresso contenente tutti i file sorgenti e avente come nome
  [java] progetto 2018_2019.zip
  l'email deve avere come oggetto
  [java] consegna progetto programmazione 2018/2019
  e deve contenere nel testo la lista (Cognome, Nome e Matricola) di tutti gli studenti che hanno svolto il progetto consegnato.
  In caso di consegna in gruppo, la consegna deve essere effettuata da uno dei componenti del gruppo, mettendo in CC nell'email anche l'indirizzo di tutti gli altri componenti.

Testo del progetto:

Sviluppare in **Java** un insieme di metodi per la gestione di insiemi e di multiinsiemi di numeri in virgola mobile. Dove indicato, i metodi devono rispettare il tempo di esecuzione richiesto.

Prima parte:

Gli insiemi devono essere gestiti in Java come array di **double**, contenente tutti i numeri presenti nell'insieme.

La libreria deve essere composta dai seguenti metodi, tutti inclusi nella stessa classe **Insiemi** salvata nel file **Insiemi.java**:

- `public static double[] rimuoviDuplicati (double [] a)`
  prende come parametro un array di double e restituisce un nuovo array di double in cui sono presenti tutti gli elementi di a senza duplicati.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**, dove n=a.length.

- `public static int cardinalita (double [] a)`
  prende come parametro un array di double e restituisce la cardinalità dell'insieme da esso rappresentato.
- `public static boolean appartiene (double [] a, double n)`
  prende come parametro un array di double ed un double n, e restituisce ***true*** se n appartiene all'insieme rappresentato da a, ***false*** altrimenti.
- `public static double [] creaSingleton (double n)`
  prende come parametro un double n, e restituisce un array che rappresenta un insieme formato dal solo elemento n.
- `public static double [] unione (double [] a, double [] b)`
  prende come parametri due array di double, e restituisce un nuovo array di double corrispondente all'insieme ottenuto facendo l'unione degli insiemi rappresentati da a e b.
  [nota: si può sfruttare il metodo `rimuoviDuplicati`]
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static double [] intersezione (double [] a, double [] b)`
  prende come parametri due array di double, e restituisce un nuovo array di double corrispondente all'insieme ottenuto facendo l'intersezione degli insiemi rappresentati da a e b.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static double [] differenza (double [] a, double [] b)`
  prende come parametri due array di double, e restituisce un nuovo array di double corrispondente all'insieme ottenuto facendo la differenza insiemistica tra l'insieme rappresentato da a e quello rappresentato da b.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static boolean sottoinsieme (double [] a, double [] b)`
  prende come parametri due array di double, e restituisce ***true*** se l'insieme rappresentato da a è un sottoinsieme di quello rappresentato da b, ***false*** altrimenti.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static boolean uguale (double [] a, double [] b)`
  prende come parametri due array di double, e restituisce ***true*** se l'insieme rappresentato da a è lo stesso di quello rappresentato da b, ***false*** altrimenti.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
  [nota: si può sfruttare il metodo `sottoinsieme`]

Seconda parte:

Un multi-insieme è una collezione finita di elementi non ordinati. Gli elementi in un multi-insieme possono essere ripetuti e si chiama **molteplicità** di un elemento x in un multi-insieme A il numero di volte in cui x compare in A.

I multiinisiemi devono essere gestiti come array di `Elem`, dove il tipo riferimento `Elem` è definito nel modo seguente:

```
class Elem{
   double valore;
   int molteplicita;
}
```

Si noti come, in tale array, non devono esistere due elementi aventi lo stesso campo valore.

La libreria deve essere composta dai seguenti metodi, tutti inclusi nella stessa classe **Multiinsiemi** salvata nel file **Multiinsiemi.java**:

- `public static Elem[] trasforma (double[] a)`
  prende come parametro un array di double contenente una collezione di elementi (con possibili ripetizioni) restituisce un nuovo array di Elem in cui ad ogni double è associata l'opportuna molteplicità. Ad esempio **a** potrebbe essere la collezione {{1.5, 6, 2, 3, 6, 6, 3, 4, 4}} contenente 1.5 e 2 con molteplicità 1; 3 e 4, con molteplicità 2; 6 con molteplicità 3.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**, dove n=a.length.
- `public static Elem[] normalizza (Elem[] a)`
  prende come parametro un array di Elem , e restituisce un nuovo array di Elem in cui non sono presenti due elementi aventi lo stesso campo valore. Se nell'array **a** erano presenti due o più elementi con lo stesso campo valore, nel nuovo array deve essere presente un solo elemento con quel campo valore, la cui molteplicità è pari alla somma delle molteplicità degli elementi che in **a** avevano lo stesso campo valore.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**, dove n=a.length.
- `public static int molteplicita (Elem[] a, double n)`
  prende come parametro un array di Elem ed un double n, e restituisce la molteplicità di n nel multiinsieme rappresentato da a. Se n non appartiene a tale multiinsieme, la sua molteplicità è 0.
- `public static Elem[] creaSingleton (double n)`
  prende come parametro un double n, e restituisce un array che rappresenta un multiinsieme formato dal solo elemento n (con molteplicità 1).
- `public static Elem[] unione (Elem[] a, Elem[] b)`
  prende come parametri due array di Elem, e restituisce un nuovo array di Elem corrispondente al multiinsieme ottenuto facendo l'unione dei multiinsiemi rappresentati da a e b. L'unione di due multi-insiemi si ottiene prendendo tutti gli elementi (comuni e non comuni) ai 2 multiinsiemi, con la **massima** molteplicità.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.

- `public static Elem[] intersezione (Elem[] a, Elem[] b)`
  prende come parametri due array di Elem, e restituisce un nuovo array di Elem corrispondente al multiinsieme ottenuto facendo l'intersezione dei multiinsiemi rappresentati da a e b. L'intersezione di due multiinsiemi si ottiene prendendo solo gli elementi comuni ai due multiinsiemi, con la **minima** molteplicità.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static Elem[] somma (Elem[] a, Elem[] b)`
  prende come parametri due array di Elem, e restituisce un nuovo array di Elem corrispondente al multiinsieme ottenuto facendo la somma dei multiinsiemi rappresentati da a e b. La somma di due multi-insiemi si ottiene prendendo tutti gli elementi (comuni e non comuni) ai 2 multiinsiemi, e **sommando** le loro molteplicità.
  [nota: si può sfruttare il metodo `normalizza`]
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static Elem[] differenza (Elem[] a, Elem[] b)`
  prende come parametri due array di Elem, e restituisce un nuovo array di Elem corrispondente al multiinsieme ottenuto facendo la differenza dei multiinsiemi rappresentati da a e b. La differenza di due multi-insiemi si ottiene prendendo tutti gli elementi del primo multiinsieme e sottraendo dalla loro molteplicità la molteplicità del corrispondente elemento (con lo stesso campo valore, se è presente) del secondo multiinsieme. Se tale differenza è minore o uguale a 0, l'elemento viene rimosso dal multiinsieme risultato.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static boolean sottoinsieme (Elem[] a, Elem[] b)`
  prende come parametri due array di Elem, e restituisce ***true*** se il multiinsieme rappresentato da a è un sottoinsieme di quello rappresentato da b (ovvero ogni elemento presente in a è presente anche in b con una molteplicità almeno uguale a quella che l'elemento ha in a), ***false*** altrimenti.
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.
- `public static boolean uguale (Elem[] a, Elem[] b)`
  prende come parametri due array di Elem, e restituisce ***true*** se il multiinsieme rappresentato da a è lo stesso di quello rappresentato da b, ***false*** altrimenti.
  [nota: si può sfruttare sfrutti il metodo `sottoinsieme`]
  Il metodo deve avere tempo di esecuzione nel caso peggiore **O(n log n)**,
  dove n=a.length+b.length.