Lab. Programmazione (CdL Informatica) & Informatica (CdL Matematica) a.a. 2022-23

Monica Nesi

Università degli Studi dell'Aquila

16 Novembre 2022

Definizione di metodi statici con array bidimensionali

I problemi visti per le sequenze di sequenze di elementi verranno ora risolti tramite metodi con array bidimensionali.

Ricordiamo che un array bidimensionale di elementi di un qualsiasi tipo T è implementato in Java come un array (monodimensionale) di array (monodimensionali) di elementi in T.

Molti problemi su strutture bidimensionali implicano l'utilizzo di più costrutti iterativi, uno dentro l'altro: quello esterno per esaminare le *righe* della struttura bidimensionale, quello interno per esaminare gli *elementi di una riga*, una volta fissata.

Quindi, nel codice dei corrispondenti metodi Java avremo tipicamente *cicli annidati* per operare su array bidimensionali.

Qui assumiamo che ogni array bidimensionale sia non vuoto.

Massimo di un array bidimensionale di interi

Scrivere un metodo che, dato un array bidimensionale di interi a, restituisce il valore massimo in a.

Input: `int[][] a`
Output: valore massimo in a

```
public static int maxArrayBi (int[][] a) {
  int max = a[0][0];
  for (int i=0; i<a.length; i++)
    for (int j=0; j<a[i].length; j++)
      if (a[i][j] > max)
        max = a[i][j];
  return max;
}
```

Altra soluzione: maxArrayBi invoca maxArray

Il metodo `maxArrayBi` potrebbe essere definito in modo diverso, utilizzando il metodo `maxArray` sulle righe `a[i]`.

```
public static int maxArrayBi (int[][] a) {
  int max = a[0][0];
  for (int i=0; i<a.length; i++) {
    int maxRiga = Max.maxArray(a[i]);
    if (maxRiga > max)
      max = maxRiga;
  }
  return max;
}
```

La variabile locale `max` potrebbe anche essere inizializzata con
`int max = Max.maxArray(a[0]);`
e poi il `for` potrebbe cominciare con `int i=1;`.

Una classe di test per maxArrayBi

Assumiamo che anche il metodo `maxArrayBi` sia nella classe `Max`.

In una classe di test `MaxTest` passare un array bidimensionale da input tramite `args` non è agevole. Scriviamo un semplice metodo `main` per testare `maxArrayBi` definendo un array bidimensionale *locale* al `main` per *enumerazione* come segue:

```
public static void main (String[] args) {
  int[][] a = {{2,-5,10,6},{23,0},{18,-3,45}};
  System.out.println("Massimo = " +
Max.maxArrayBi(a));
}
```

Dopo la compilazione con `javac MaxTest.java`, si ha:

```
java MaxTest
Massimo = 45
```

Contare occorrenze in un array bidimensionale

Es.I. Scrivere un metodo che, dati un array bidimensionale di interi a ed un intero x, restituisce il numero delle occorrenze di x in a.

Input: `int[][] a, int x`
Output: numero di occorrenze di x in a

```
public static int occorrenzeBi (int[][] a,
int x) {
  int cont = 0;
  for (int i=0; i<a.length; i++)
    for (int j=0; j<a[i].length; j++)
      if (a[i][j] == x)
        cont++;
  return cont;
}
```

Contare occorrenze: un'altra soluzione

Di nuovo, avendo un metodo `occorrenze` (definito nella classe `ArrayMono`) che calcola le occorrenze di un elemento all'interno di un array monodimensionale, si potrebbe definire il metodo `occorrenzeBi` in modo da invocare `occorrenze` sulle righe `a[i]` (*programmazione strutturata e modulare*).

```
public static int occorrenzeBi (int[][] a,
int x) {
  int cont = 0;
  for (int i=0; i<a.length; i++)
    cont += ArrayMono.occorrenze(a[i],x);
  return cont;
}
```

In modo simile possiamo contare quanti elementi di un array bidimensionale soddisfano una data proprietà.

Sommare i numeri in un array bidimensionale

Scrivere un metodo che, dato un array bidimensionale di interi a, restituisce la somma degli elementi in a.

Input: `int[][] a`
Output: sommatoria degli elementi in a

```
public static int sommaABi (int[][] a) {
  int sum = 0;
  for (int i=0; i<a.length; i++)
    for (int j=0; j<a[i].length; j++)
      sum = sum + a[i][j];
  return sum;
}
```

È anche possibile rimpiazzare il ciclo interno invocando il metodo `sommaA` sulle righe `a[i]` in modo simile a quanto fatto in `occorrenzeBi` con il metodo `occorrenze`.

Verificare l'occorrenza di un elemento

Es.II. Scrivere un metodo che, dati un array bidimensionale di interi a ed un intero x, restituisce true se x occorre in a, false altrimenti.

Input: int[][] a, int x
Output: true se x in a, false altrimenti

```
public static boolean occorreBi (int[][] a,
int x) {
  for (int i=0; i<a.length; i++)
    for (int j=0; j<a[i].length; j++)
      if (a[i][j] == x)
        return true;
  return false;
}
```

Questa è la soluzione con return senza la variabile booleana trovato (vedi diagramma di flusso).

Tutti gli elementi con una data proprietà

Scrivere un metodo che, dato un array bidimensionale di interi `a` ed un intero `k`, restituisce `true` se *tutti* gli elementi di `a` sono multipli di `k`, `false` altrimenti.

Input: `int[][] a`, `int k`
Output: `true` se *ogni* `a[i][j]` è multiplo di `k`, `false` altrimenti

Tutti gli elementi con una data proprietà (cont.)

```
public static boolean tuttiMultipli(int[][] a,
int k) {
  int i = 0;
  while (i<a.length) {
    int j = 0;
    while (j<a[i].length)
      if (a[i][j] % k == 0)
        j++;
      else
        return false;
    i++;
  }
  return true;
}
```

Tutti gli elementi con una data proprietà (cont.)

I due comandi `while` annidati possono essere sostituiti da due comandi `for` annidati:

```
public static boolean tuttiMultipli(int[][] a,
int k) {
  for (int i=0; i<a.length; i++)
    for (int j=0; j<a[i].length; j++)
      if (a[i][j] % k != 0)
        return false;
  return true;
}
```

La condizione verificata nel comando condizionale è opposta a quella data, in modo da non avere comandi vuoti.

È anche possibile utilizzare una variabile booleana, e.g. `ok`, inizializzata a `true`, che viene messa a `false` *non appena* si trova il primo elemento che non soddisfa la proprietà richiesta.

Verificare l'occorrenza di un elemento in ogni riga

Es.III. Scrivere un metodo che, dati un array bidimensionale di interi `a` ed un intero `x`, restituisce `true` se `x` occorre in *ogni riga* di `a`, `false` altrimenti.

Input: `int[][] a`, `int x`
Output: `true` se `x` in *ogni* `a[i]`, `false` altrimenti

Si tratta di una proprietà *locale* da verificare.

Non appena si trova un'occorrenza di `x` nella riga in esame, si deve passare alla riga successiva.

Non appena si trova una riga che non contiene alcuna occorrenza di `x` (i.e., tale riga non soddisfa la condizione richiesta), allora si deve terminare il metodo restituendo `false`.

Occorrenza di un elemento in ogni riga: una soluzione

```
public static boolean occorreBiOgniRiga
(int[][] a, int x) {
  int i = 0;
  while (i<a.length) {
    int j = 0;
    boolean trovato = false;
    while (j<a[i].length && !trovato) {
      if (a[i][j] == x)
        trovato = true;
      else
        j++;
    }
    if (!trovato) return false;
    i++;
  }
  return true;
}
```

Occorrenza di un elemento in ogni riga: una soluzione (cont.)

La variabile booleana `trovato` serve per passare alla riga successiva *non appena* si trova un'occorrenza di `x` nella riga in esame `a[i]`.

La variabile booleana `ok` (vedi diagramma di flusso) è stata rimpiazzata da un opportuno utilizzo di `return`.

Almeno k elementi con una data proprietà in ogni riga

Es.IV. Scrivere un metodo che, dati un array bidimensionale di interi `a` ed un intero `k`, restituisce `true` se in *ogni riga* `a[i]` di `a` esistono *almeno* `k` numeri strettamente positivi, `false` altrimenti. Si assuma `k>0`.

Input: `int[][] a`, `int k`
Output: `true` se in *ogni* `a[i]` esistono *almeno* `k` numeri strettamente positivi, `false` altrimenti

Almeno k elementi con una data proprietà in ogni riga: una soluzione

```
public static boolean almenokPosBi(int[][] a,
int k) {
  for (int i=0; i<a.length; i++) {
    int j = 0, cont = 0;
    while (j<a[i].length && cont<k) {
      if (a[i][j] > 0)
        cont++;
      j++;
    }
    if (cont<k)
      return false;
  }
  return true;
}
```

Almeno k elementi con una data proprietà in ogni riga: una soluzione (cont.)

Di nuovo, la variabile booleana `ok` (vedi diagramma di flusso) è stata rimpiazzata da un opportuno utilizzo di `return`.

Come definire questo metodo invocando il metodo `almenokPos` che opera su array monodimensionali?

Alcuni esercizi

Esercizio 1 (Secondo Parziale - 13 Dicembre 2021)

Scrivere un metodo *statico* iterativo che, dati un array bidimensionale di stringhe a, un array monodimensionale di caratteri c ed un intero k, restituisce *true* se in *ogni* riga $a[i]$ *esiste almeno* una stringa il cui primo carattere è uguale a $c[i]$ e la cui lunghezza è minore di k; altrimenti il metodo restituisce *false*.
Esempio: se $a =$ `{{"abc","bb","d"},{"rkr","rc"},{"mp","aqsfg","ats","ar"}}`,
$c =$ `{'b','r','a'}` e $k = 4$, il metodo restituisce *true*.
Si assuma che gli array a e c abbiano lunghezza uguale, che ogni stringa in a sia non vuota e $k > 0$.

N.B. i) Occorre passare alla riga successiva *non appena* si trova in $a[i]$ la prima stringa che soddisfa la condizione data.
ii) Occorre restituire *false non appena* si trova la prima riga $a[i]$ che non soddisfa la condizione richiesta.

Alcuni esercizi (cont.)

Esercizio 2 (Secondo Parziale - 30 Novembre 2018)

Scrivere un metodo *statico* iterativo che, dati un array bidimensionale di interi a, un array monodimensionale di interi b ed un intero k, restituisce *true* se in *ogni* riga $a[i]$ di a esistono *almeno* k numeri minori o uguali a $b[i]$, altrimenti il metodo restituisce *false*.
Esempio: se $a =$ `{{7,3,-2,2,9},{5,0,8,3},{15,11,-2}}`, $b =$ `{6,3,11}` e $k = 2$, il metodo restituisce *true*.

N.B. i) Occorre restituire *false* non appena si trova una riga $a[i]$ che non soddisfa la condizione richiesta.
ii) Occorre passare alla riga successiva non appena si verifica che una riga $a[i]$ soddisfa la condizione richiesta.
iii) Si assuma che gli array a e b abbiano la stessa lunghezza e $k > 0$.