Università di Torino – Facoltà di Scienze MFN
Corso di Studi in Informatica

Programmazione I - corso B
a.a. 2009-10

prof. Viviana Bono

Blocco 14 – Array parzialmente riempiti. Algoritmi sugli array.

Array parzialmente riempiti

Un oggetto array, una volta creato, ha una lunghezza fissa che non può più essere variata. Spesso, però, il numero di elementi di un insieme o di una sequenza (ad esempio il numero dei libri di una biblioteca, oppure degli articoli in vendita in un negozio, ecc.) deve poter cambiare nel tempo, con l'aggiunta e la rimozione di elementi.

Un array di lunghezza virtuale "variabile" viene realizzato allocando un array di una certa lunghezza corrispondente al numero massimo di elementi ipotizzabile per quella applicazione, e poi riempiendolo solo parzialmente, cominciando dall'elemento di indice 0 e tenendo gli elementi occupati sempre tutti consecutivi (cioè la parte occupata dell'array deve essere una parte iniziale e priva di "buchi").

Naturalmente occorre tenere in una variabile numElementi la lunghezza della parte occupata.

Per realizzare ciò in modo coerente, si definisce una classe di cui ciascun oggetto contiene un campo **elementi** di tipo (riferimento ad) array, ed un campo **numElementi** di tipo intero che contiene la lunghezza della parte occupata. Tali campi sono dichiarati privati, in modo che l'array vero e proprio e il campo numElementi possano essere modificati solo da opportuni metodi della classe, che devono garantire che numElementi sia sempre il numero di elementi effettivi presenti nell'array.

Array parzialmente riempiti: definizione della classe

```
public class IntArrayParziale {
  private int[] elementi;
  private int numElementi;
  ...
```

in ciascun oggetto della classe:

il campo **elementi** conterrà il riferimento ad un array ordinario, che viene creato all'istante della creazione dell'oggetto stesso, quindi all'interno del costruttore; la lunghezza desiderata per l'array verrà fornita come argomento;

il il campo **numElementi** conterrà ad ogni istante il numero di elementi effettivamente presenti (dal punto di vista logico) nell'array, cioè la lunghezza della parte occupata; si noti che in tal modo **numElementi** è l'indice del primo posto libero (se esiste).

Il costruttore è quindi:

```
   public IntArrayParziale(int lunghezzaMax) {
    elementi = new int[lunghezzaMax];
    numElementi = 0;
  }
```

Si noti che l'oggetto viene creato vuoto, cioè con **numElementi** = 0 (naturalmente, nel caso di un array di interi, ogni cella dell'array elementi non è mai "vuota" ma contiene sempre un intero, che tuttavia per l'utente è logicamente inesistente).

Array parzialmente riempiti: definizione della classe

ATTENZIONE: Non confondere:

elementi.length, che è la lunghezza fisica dell'array allocato, e rappresenta quindi la *capacità*, ossia il numero massimo di valori che l'oggetto può contenere;

e

numElementi, che è la dimensione logica dell' array, cioè la lunghezza della parte occupata; naturalmente deve sempre essere:

numElementi <= elementi.length

All'array elementi e al campo numElementi di un oggetto si può accedere e li si può modificare soltanto richiamando su quell'oggetto i metodi della classe.

Avremo perciò almeno due metodi: uno per aggiungere un elemento ed uno che restituisca l'elemento di indice i; inoltre vorremo avere un metodo che permetta di modificare o di rimuovere l'elemento di indice i; ecc.

NOTA BENE: nessuno di questi metodi deve fare input o output; l'input-output deve essere riservato a metodi appositi, che richiameranno i metodi precedenti (in realtà, in programmi un po' più realistici i metodi di input-output vengono definiti addirittura in una classe diversa, ad es. di nome InterfacciaUtente).

Il nucleo del metodo che aggiunge un elemento in ultima posizione è:

```
public void aggiungiElemento(int val) {
  elementi[numElementi] = val;
  numElementi++;
}
```

ma in questo modo non si controlla che l'array non sia già pieno; una possibile semplice soluzione è quindi:

```
public boolean aggiungiElemento(int val) {
  if(numElementi == elementi.length)
    return false     // l'operazione non e` andata a buon fine
  else {
  elementi[numElementi] = val;
  numElementi++;
  return true;       // l'operazione e` andata a buon fine
  }
}
```

Una soluzione migliore si ottiene definendo il metodo in modo che, quando si vuole inserire un nuovo elemento in un array pieno, si crea prima un array di dimensione maggiore (ad es. doppia) e si ricopiano in esso tutti gli elementi dell'array originale, che viene rimpiazzato dal nuovo array.

Array ordinari (cioè "normali") e array parzialmente riempiti

Sarebbe utile poter trasformare un ordinario (cioè normale) array pieno in un oggetto della classe IntArrayParziale, e viceversa. A tal fine definiamo:

- un costruttore IntArrayParziale(int[] a) che prende come argomento un array pieno e costruisce un corrispondente oggetto della classe IntArrayParziale;
- un metodo int[] toArray() che costruisce e restituisce un array pieno contenente tutti gli elementi presenti nella parte occupata dell'array parziale.

Attenzione! La seguente semplice versione del costruttore precedente è scorretta:

```
public IntArrayParziale(int[] elementi) {
   this.elementi = elementi;
   numElementi = elementi.length;
 }
```

Essa infatti rende accessibile pubblicamente, attraverso l'aliasing, l'array **elementi** che invece, essendo privato, dovrebbe essere accessibile soltanto attraverso i metodi dell'oggetto.

Si considerino ad esempio le seguenti istruzioni (magari inserite in un main di prova):

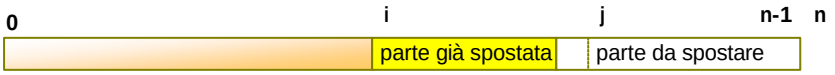
```
int[] unArray = {13, 21, -5, 43, 8};
IntArrayParziale arrParz = new IntArrayParziale(unArray);
unArray[2] = 100;
```

l'ultima istruzione modifica l'array privato elementi contenuto in arrParz senza usare i metodi dell'oggetto stesso. **Come fare?**

Array parz. riempito: eliminazione dell'elemento di indice i

Problema: eliminare l'elemento di indice i, compattando l'array in modo da non lasciare il posto libero, mantenendo gli elementi nello stesso ordine, senza usare array ausiliari.

INVARIANTE



la porzione di array fra gli indici i e j-2 (inclusi) è già nei posti giusti;
j-1 è l'indice del posto libero;
la porzione di array fra gl'indici j e n-1 (inclusi) è ancora da spostare.

PASSO GENERICO

```
elementi[j-1] = elementi[j];
```



```
public boolean eliminaElementoDiIndice(int i)
{
  if(i < 0 || i >= numElementi) {
    return false // l'op. non e` possibile
  }
  for(int j = i+1; j < numElementi; j++) {
    elementi[j-1] = elementi[j];
  }
  numElementi--;
  return true; // l'op. e` stata eseguita
}
```

Versione con un solo indice

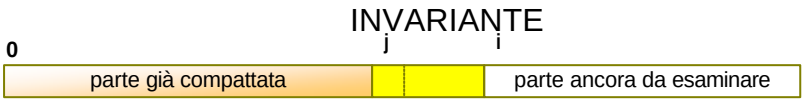
In realtà, poiché il valore dell'indice i, passato come argomento, non viene più utilizzato dopo la fine del ciclo for, i potrebbe essere usato direttamente come indice corrente nel ciclo, facendo a meno di j :

```
public boolean eliminaElementoDiIndice(int i)
{
  if(i < 0 || i >= numElementi) {
    return false;
  }
  for( ; i < numElementi-1; i++) {
    elementi[i] = elementi[i+1];
  }
  numElementi--;
  return true;
}
```

Molti tuttavia preferiscono uno stile di programmazione in cui gli argomenti di un metodo, se sono di tipi primitivi (int, double, ecc.), non vengono modificati.

Array parz. riempito: eliminazione di tutti gli elementi uguali a x

Problema: eliminare tutti gli elementi uguali a un dato valore, compattando l'array in modo da non lasciare buchi, mantenendo gli elementi nello stesso ordine, senza usare array ausiliari.



Array parz. riempito: eliminazione di tutti gli elementi uguali a x

Problema: eliminare tutti gli elementi uguali a un dato valore, compattando l'array in modo da non lasciare buchi, mantenendo gli elementi nello stesso ordine, senza usare array ausiliari.

INVARIANTE

Passo generico

Esamino il primo elemento della parte ancora da esaminare, **a[i]** :

se è uguale a x, non lo copio nella parte compattata finale;

se non lo è, allora deve rimanere nell'array risultato, quindi lo copio nella parte già compattata, incrementando poi la lunghezza di tale parte: `a[j] = a[i]; j++`

In entrambi i casi avanzo di 1 nella parte da esaminare: `i++`


```
public void eliminaTutti(int x) {
    int i = 0, j = 0;
    while(i < numElementi) {
      if(elementi[i] != x) {
        elementi[j] = elementi[i];
        j++;
      }
      i++;
    }
    numElementi = j;
  }
```

Versione con il for

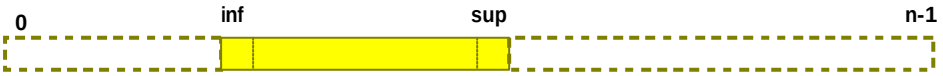
Osserviamo che nel metodo precedente si ha un ciclo while controllato dalla condizione i < numElementi, dove il contatore i viene inizializzato a 0 e poi incrementato di 1 ad ogni passo. Un tale ciclo può essere espresso in modo più conciso, ma ugualmente chiaro, per mezzo di un for:

```
public void eliminaTutti(int x) {
  int j = 0;
  for(int i = 0; i < numElementi; i++) {
    if(elementi[i] != x) {
      elementi[j] = elementi[i];
      j++;
    }
  }
  numElementi = j;
}
```


Ricerca Binaria

Ricerca binaria in array (pieno) ordinato

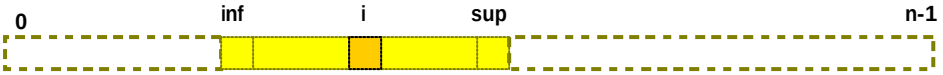
INVARIANTE



Il valore **x** da cercare, se è presente nell'array, si trova nella porzione di array compresa fra gl'indici **inf** e **sup** (inclusi).

Ricerca binaria in array ordinato

INVARIANTE



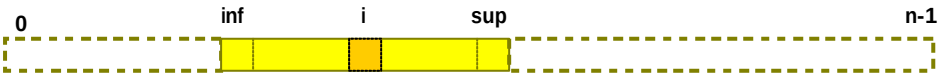
Il valore **x** da cercare, se è presente nell'array, si trova nella porzione di array compresa fra gl'indici **inf** e **sup** (inclusi).

PASSO GENERICO

Calcolo il valore dell'indice **i** dell'elemento centrale della porzione di array; confronto **x** con tale elemento **`a[i]`**; sono possibili tre casi:

Ricerca binaria in array ordinato

INVARIANTE



Il valore `x` da cercare, se è presente nell'array, si trova nella porzione di array compresa fra gl'indici **inf** e **sup** (inclusi).

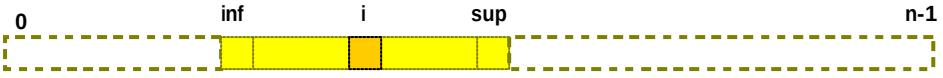
PASSO GENERICO

Calcolo il valore dell'indice `i` dell'elemento centrale della porzione di array; confronto `x` con tale elemento `a[i]`; sono possibili tre casi:

- `x < a[i]` x, se c'è, si trova nella porzione compresa fra inf e i-1 (inclusi);

Ricerca binaria in array ordinato

INVARIANTE



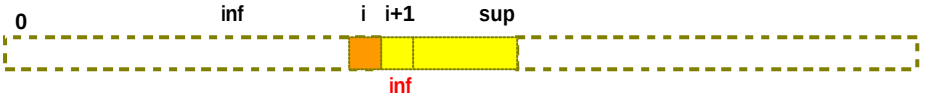
Il valore `x` da cercare, se è presente nell'array, si trova nella porzione di array compresa fra gl'indici **inf** e **sup** (inclusi).

PASSO GENERICO

Calcolo il valore dell'indice `i` dell'elemento centrale della porzione di array; confronto `x` con tale elemento `a[i]`; sono possibili tre casi:

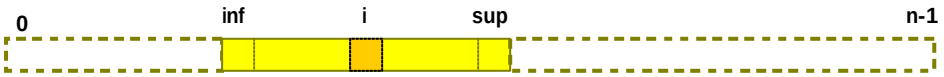
- `x < a[i]` x, se c'è, si trova nella porzione compresa fra inf e i-1 (inclusi);

- `x > a[i]` x, se c'è, si trova nella porzione compresa fra i+1 e sup (inclusi);



Ricerca binaria in array ordinato

- `x == a[i]` ho trovato il valore cercato, quindi termino il metodo restituendo true oppure l' indice i



Qual è la condizione per cui il ciclo deve continuare?

La porzione di array su cui fare la ricerca non deve essere vuota; cioè

inf <= sup

NOTA: se inf == sup la porzione non è vuota, ma contiene un elemento.

Quali sono i valori iniziali di inf e sup ?

Inizialmente la porzione di array su cui effettuare la ricerca è l'intero array.

inf = 0; sup = a.length - 1;

Ricerca binaria con risultato booleano

```
public static boolean ricercaBin(int x, int[] a){
  int i, inf = 0, sup = a.length - 1;
  while(inf <= sup) {
    i = (inf + sup)/2;
    if(x < a[i]) sup = i-1;
    else if(x > a[i]) inf = i+1;
    else return true;
  }
  return false;
}
```


Ricerca binaria con restituzione dell'indice

```
public static int ricercaBin(int x, int[] a) {
  int i, inf = 0, sup = a.length - 1;
  while(inf <= sup) {
    i = (inf + sup)/2;
    if(x < a[i]) sup = i-1;
    else if(x > a[i]) inf = i+1;
    else return i;
  }
  return -1;
}
```

Complessità computazionale della ricerca binaria

Ad ogni passo la porzione di array su cui effettuare la ricerca si divide per 2; se n è la lunghezza dell'array, il numero di iterazioni del ciclo è, nel caso peggiore, circa $\log_2 n$; ad esempio, se n = 1024, al secondo passo la porzione su cui ricercare sarà di dimensione 512, al terzo 256, e così via; dopo 10 passi si sarà ridotta a 1, poiché $1024 = 2^{10}$.

Tale caso peggiore si ha quando il valore da cercare non esiste, oppure quando viene trovato all'ultimo passo.

Raffinamento

Se il valore da ricercare è minore del primo elemento o maggiore dell'ultimo il metodo precedente compie $\log_2 n$ passi (dove n è la lunghezza dell'array).

È facile e conveniente ottimizzare questi due casi introducendo un test prima del ciclo.

```
public static int ricercaBin(int x, int[] a) {
  int inf = 0, sup = a.length - 1;
  if(x < a[0] || x > a[sup]) return -1;
  while(inf <= sup) {
    int i = (inf + sup)/2;
    if(x < a[i]) sup = i-1;
    else if(x > a[i]) inf = i+1;
    else return i;
  }
  return -1;
}
```

Versione finale

Infine osserviamo che entrambe le versioni precedenti della ricerca binaria generano un errore se l'array è vuoto, cioè se ha lunghezza 0, il che in Java è possibile. Naturalmente se l'array è vuoto il valore cercato sicuramente non c'è, quindi inseriamo una ulteriore condizione fra quelle per le quali il risultato è –1 (osservando che se a.length è 0, allora sup è –1):

```
public static int ricercaBin(int x, int[] a) {
    int inf = 0, sup = a.length - 1;
    if(sup == -1 || x < a[0] || x > a[sup]) return -1;
    while(inf <= sup) {
      int i = (inf + sup)/2;
      if(x < a[i]) sup = i-1;
      else if(x > a[i]) inf = i+1;
      else return i;
    }
    return -1;
  }
```

si noti che se l'espr. bool. **sup==-1** è true le condizioni successive non vengono valutate e quindi non si generano errori runtime.

Algoritmi piu' avanzati

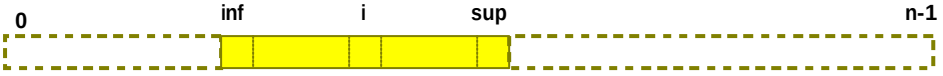
Ricerca binaria: versione che restituisce l'indice del punto di inserimento.

In un array ordinato in cui siano ammessi elementi ripetuti, l'inserimento di un nuovo elemento uguale ad uno già presente può essere effettuato esattamente al posto di quello già presente, che insieme a tutti i successivi verrà preventivamente spostato a destra di un posto.

Nel caso di valore già presente nell'array l'indice restituito dal metodo di ricerca binaria è perciò proprio il punto di inserimento.

Cerchiamo ora di definire una versione di tale metodo la quale anche nel caso (più interessante) di valore non già presente restituisca il punto di inserimento (invece di –1).

Ricerca binaria: versione che quando il valore cercato non è già presente restituisce l'indice del punto di inserimento.



Riscriviamo l'invariante nel modo seguente (dove x sia il valore da cercare):

a[0] <= ... a[inf-2] <= a[inf-1] < x < a[sup+1] <= a[sup+2] <= ... <= a[n-1]

Se x non è presente nell'array, all'uscita dal while si ha ancora:

a[inf-1] < x < a[sup+1] e inoltre sup = inf –1

Quindi:

a[inf-1] < x < a[inf]

Pertanto il valore x, nel caso se ne richieda l'inserimento (ordinato) nell'array, dovrebbe essere inserito nell'elemento di indice inf, naturalmente spostando di una posizione a destra tutti gli elementi da a[inf] (compreso) fino ad a[n-1].

Se x non è presente nell'array, il metodo deve quindi restituire inf. Alternativamente, si può far restituire al metodo un valore negativo da cui sia possibile ricavare inf, cioè:

– inf –1

quest'ultima è la soluzione scelta dalla Sun per la libreria di java.

Ricerca binaria: versione Sun che, se il valore cercato non è presente, restituisce –1 – indice_del_punto_di_inserimento

```
public static int binarySearch(int[] a, int x) {
  int inf = 0;
  int sup = a.length-1;
  if(sup == -1 || x < a[0]) return -1;
  if(x > a[sup]) return -a.length - 1;
  while (inf <= sup) {
    int i = (inf + sup)/2;
    if(x < a[i]) sup = i-1;
    else if(x > a[i]) inf = i+1;
    else return i;
  }
  return -inf - 1;  // key not found.
}
```

ovviamente l'inserimento in un array pieno non è possibile, quindi il metodo di cui sopra non è molto utile ...; scriviamone allora una versione per array parzialmente riempito (vedi pagine successive).

Ricerca binaria in array parzialmente riempito: versione che, se il valore cercato non è presente, restituisce il punto di inserimento

Si immagini di definire una classe IntArrayOrdinato, simile ad IntArrayParziale, ma in cui gli elementi siano mantenuti in ordine.

```
public class IntArrayOrdinato {
  ...
  public int ricercaBinaria(int x) {
    int inf = 0, sup = numElementi - 1;
    if(sup == -1 || x < elementi[0]) return 0;
    if(x > elementi[sup]) return sup+1;
    while(inf <= sup) {
      int i = (inf + sup)/2;
      if(x < elementi[i]) sup = i-1;
      else if(x > elementi[i]) inf = i+1;
      else return i;
    }
    return inf;
  }
 ...
```

Versione "alla Sun" (se il valore è assente restituisce – inf – 1)

```
public class IntArrayOrdinato {
  ...
  public int ricercaBinaria(int x) {
    int inf = 0, sup = numElementi - 1;
    if(sup == -1 || x < elementi[0]) return -1;
    if(x > elementi[sup]) return –(numElementi+1);
    while(inf <= sup) {
      int i = (inf + sup)/2;
      if(x < elementi[i]) sup = i-1;
      else if(x > elementi[i]) inf = i+1;
      else return i;
    }
    return –(inf+1);
  }
 ...
```


Nota

Per inserire un nuovo elemento in un array ordinato di lunghezza n si può usare la versione precedente della ricerca binaria che restituisce il punto di inserimento in un tempo medio proporzionale a log n, invece che proporzionale ad n come nella ricerca sequenziale. Tuttavia lo spostamento di tutti gli elementi successivi (per far posto al nuovo) richiede comunque in media un numero di passi proporzionale ad n.

Anche usando la ricerca binaria, il tempo necessario per effettuare l'inserimento in un array ordinato rimane quindi lineare in n.