lezione 3

Correttezza di algoritmi iterativi (invarianti)

Esercizio: Stampa

Problema: scrivere un algoritmo che, dato un array A, stampa tutti i suoi elementi

Stampa(A)

1. i ← 1
2. for i ← 1 to length(A)
3. ▶ INV
4. do stampa A[i]

INV: tutti gli elementi di A[1…i-1] sono stati stampati

Inizializzazione: A[1..0] è vuoto

Mantenimento: sia vero per i fissato. Quindi tutti gli elementi di A[1..i-1] sono stati stampati. Il corpo del ciclo stampa A[i]. Quindi tutti gli elementi di A[1..i] sono stati stampati, ovvero l'invariante viene mantenuto all'incrementare di i.

Terminazione: i= length(A)+1 e l'invariante garantisce che tutti gli elementi di A[1..length(A)] sono stati stampati.

Esercizio: Primi

Problema: dato un array A, scrivere un algoritmo che ritorna true se A contiene tutti numeri primi; ritorna false altrimenti.

Primi(A)

```
1.  i ← 1
2.  controlla ← true
3.  while i <= length(A) and controlla
4.      ▶ INV
5.      do if A[i] è primo
6.          then i ← i+1
7.          else controlla ← false
8.  return controlla
```

INV: A[1…i-1] contiene tutti numeri primi

Esercizio: Primi

INV: A[1…i-1] contiene tutti numeri primi

Inizializzazione: A[1..0] è vuoto. Vero.
Mantenimento: sia vero per i fissato. Allora A[1..i-1] contiene tutti numeri primi. Dobbiamo dimostrare che l'invariante viene mantenuto al ciclo successivo. Il corpo del ciclo esegue un if:
-se A[i] è primo allora è vero che A[1..i] contiene tutti numeri primi. In tal caso i viene incrementato e questo rende l'invariante vero per il ciclo successivo.
-se A[i] non è primo allora controlla=false e i non viene incrementato. Il ciclo termina.
Terminazione: il ciclo termina per due ragioni:
-controlla=false e i<=length(A). In tal caso è vero che A[1..i-1] contiene tutti numeri primi ma A[i] non è primo
-controlla=true e i=length(A)+1. In tal caso è vero che A[1..length(A)] contiene tutti numeri primi.
L'algoritmo ritorna correttamente il valore di controlla.

Esercizio: Indice

Problema: dato un array A, scrivere un ciclo che assegna il valore i ad A[i].

Indice(A)
1. for i ← 1 to *length*(A)
2. ▶ INV
3. do A[i] ← i

INV: per ogni j=1…i-1 si ha A[j]=j
Inizializzazione: per ogni j=1..0 si ha A[j]=j. Vero.
Mantenimento: sia vero per i fissato. Allora A[j]=j per ogni j=1..i-1.
Il corpo del ciclo esegue A[i] ← i e quindi dopo l'assegnamento si ha A[j]=j per ogni j=1..i, ovvero l'invariante viene mantenuto all'incremento di i.
Terminazione: alla fine del ciclo i= lenght(A)+1 e quindi per ogni j=1..length(A) si ha A[j]=j.

Esercizio Pari

Problema: dato un array A contenente n numeri interi, copiare i "pari" nella parte iniziale di A e ritornare la lunghezza del sottoarray di pari ottenuto.

Pari(A)
1. k ← 0
2. for i ← 1 to *length*(A)
3. ▶ INV
4. do if A[i] è pari
5. then k ←k+1
6. A[k] ← A[i]
7. return k

Esercizio Pari: correttezza

INV = A[1…k] contiene tutti e soli i numeri pari originariamente presenti in A[1…i-1]

Inizializzazione: A[1…0] è vuoto
Mantenimento: Sia vero per i fissato. Allora A[1…k] contiene tutti e soli i numeri pari che erano originariamente in A[1…i-1].
A questo punto, se A[i] è pari, k viene incrementato e A[i] viene copiato in A[k]. Se invece A[i] è dispari non si fa nulla.
In ogni caso, A[1…k] contiene tutti e soli i numeri pari che erano originariamente in A[1…i]. Quindi l'invariante viene mantenuto all'incremento di i.
Terminazione: Al termine del ciclo i = n+1, A[1…k] contiene tutti e soli i numeri pari inizialmente in A[1…n]

Esercizio: trovaIndice

Sia A un array di dimensione n contenente numeri interi positivi e negativi tali che A[1] < A[2] < …< A[n]. Scrivere un algoritmo **trovaIndice** che trova e restituisce un indice i tale che A[i] = i. Se tale indice non esiste l'algoritmo restituisce -1.

Soluzione iterativa trovaIndice

Scriviamo prima una versione iterativa:

```
trovaIndice(A)
  inf ← 1
  sup ← length(A)
  indice ← -1
  while inf ≤ sup and indice = -1
  ▶ INV
  do mid ← (inf + sup) div 2
      if A[mid] = mid
        then indice ← mid
        else if A[mid] < mid
               then sup ← mid -1
               else inf ← mid +1
  return indice
```

Correttezza trovaIndice

INV: l'elemento cercato, se esiste, si trova in A[inf,sup].

Inizializzazione: all'inizio inf=1 e sup=length(A). E' banalmente vero che l'elemento cercato, se esiste, si trova in A[1,length(A)].
Mantenimento: sia vero per inf e sup fissati. Allora è vero che l'elemento cercato, se esiste, si trova in A[inf,sup]. Dobbiamo dimostrare che l'invariante viene mantenuto al ciclo successivo.
corpo del ciclo esegue un if:
-se A[mid]=mid allora l'elemento è stato trovato e mid viene memorizzato nella variabile indice
-se A[mid]<mid allora l'elemento cercato non può essere presente in A[inf,mid] e l'algoritmo pone inf=mid+1. Quindi è vero che l'elemento cercato, se esiste, si trova in A[inf,sup], ovvero l'invariante viene mantenuto per il ciclo successivo.
-Se A[mid]>mid … il ragionamento è analogo al caso precedente.

Correttezza trovaIndice

INV: l'elemento cercato, se esiste, si trova in A[inf,sup].

Terminazione: il ciclo termina per due ragioni:
-inf>sup : in tal caso l'indice i per cui A[i]=i non è stato trovato e l'algoritmo ritorna correttamente il valore di indice, che è rimasto -1
-indice != -1 : in tal caso l'algoritmo ritorna correttamente il valore di indice
In entrambi i casi rimane vero che l'indice cercato, se esiste, si trova in A[inf,sup]

Prima esercitazione

Definiamo una collezione come un gruppo di oggetti, detti gli elementi della collezione. Tra le proprietà che caratterizzano una collezione ci sono le seguenti:
- elementi duplicati / non duplicati
- elementi ordinati / non ordinati

Ad esempio:
-un insieme è una collezione in cui gli elementi non possono essere duplicati e non devono essere necessariamente ordinati;
- un multi-insieme è una collezione in cui viene ammessa la presenza di più copie dello stesso elemento e in cui non è richiesto l'ordinamento degli elementi;
- una sequenza ordinata è una collezione in cui gli elementi compaiono in modo ordinato e sono ammesse più copie dello stesso elemento.
- insieme: {a, z, d, f, b}
- multi-insieme: {a, z, d, z, f, g, g, h, a}
- sequenza ordinata: <a, a, d, f, g, g, h, z, z>

Prima esercitazione

Le operazioni principali che consentono di operare sugli elementi di una collezione sono:

•Inserimento di un elemento (insert)
•Cancellazione di un elemento (remove)
•Verifica la presenza di un elemento nella collezione (contains)

Le operazioni di inserimento e cancellazione devono rispettare le proprietà della collezione quali l'ammissione o meno di elementi duplicati e il mantenimento o meno dell'ordinamento tra gli elementi.

Classe MultiInsieme

Vogliamo realizzare una classe MultiInsieme per rappresentare un multi insieme di elementi di tipo stringa. Si tratta quindi di rappresentare una collezione di stringhe in cui sono ammessi elementi duplicati.

La struttura dati che utilizziamo per memorizzare il multi insieme è l'array.
Poiché gli elementi di un multi insieme non devono essere necessariamente ordinati, scegliamo di non mantenere l'ordinamento all'interno dell'array che memorizza gli elementi del multi insieme.

Vogliamo inoltre utilizzare la tecnica dell'invariante per dimostrare la correttezza dei metodi della classe MultiInsieme.

Classe MultiInsieme

```
public class MultiInsieme {
    private static final int defaultSize = 50;
    private String[] M;      // il multi-insieme e' un array
    private int count;       // tot. elementi del multi-insieme
```

La classe ha campi privati, cioè non accessibili dall'esterno

```
// post: costruisce un array di dimensione defaultSize
    public MultiInsieme() {…}

    // post: ritorna il numero di elementi del multi-insieme
    public int size() {…}

    // post: ritorna true sse il multi-insieme e' vuoto
    public boolean isEmpty() {…}

    // post: svuota il multi-insieme
    public void clear() {…}
```

Classe Classe MultiInsieme

```
    // pre:  s non nulla
    // post: aggiunge la stringa s al multi-insieme ponendola in
    //       coda all'array. Ritorna true se l'operazione e'
    // riuscita, false se non c'e' piu' spazio libero nell'array
    public boolean insert(String s) {…}

    // pre:  s non nulla
    // post: ritorna true se nel multi insieme c'e' un elemento
    // uguale a s, false altrimenti.
    public boolean contains(String s) {…}

    // pre: s non nulla
    // post: ritorna il numero di occorrenze di s nel multi
    //       insieme
    public int occurrences(String s) {…}

    // post: ritorna il numero di elementi distinti del multi
    //       insieme
    public int distinct() {…}
```

Classe MultiInsieme

```
    // pre:  s non nulla
    // post: rimuove la prima occorrenza di s nell'array.
    //       Ritorna true se l'operazione e' riuscita, false
    //        altrimenti (se s non e' presente nell'array).
    public boolean remove(String s) {…}


    // post: ritorna una stringa contenente gli elementi del
    //       multi insieme. Utilizza uno spazio come separatore
    //       tra gli elementi
    public String toString() {…}
}
```

Prima esercitazione

Si richiede di:
-completare l'implementazione della classe MultiInsieme (usare i commenti Javadoc)
-dimostrare la correttezza dei metodi *contains*, *occurrences*, *distinct* e *toString* usando la tecnica dell'invariante

Viene fornita la classe TestMultiInsieme per testare tutti i metodi della classe MultiInsieme.

Consegnare entro domenica 26 Ottobre 2008