Corso di Laurea Ingegneria Informatica

Fondamenti di Informatica 1

Dispensa E07

Iterazione

C. Limongelli

Novembre 2007

Contenuti

- **Si vogliono formalizzare diverse tecniche per la risoluzione di problemi di:**
  - **Accumulazione**
  - **Conteggio**
  - **Verifica esistenziale**
  - **Verifica universale**
  - **Ricerca**

Accumulazione

- **Calcolare una proprietà sintetica (cumulata) da una sequenza di valori**
  - calcolo della somma di una sequenza di valori numerici
  - calcolo del fattoriale di un numero naturale

- **Tecnica dell'accumulazione**
  - una variabile di controllo (**accumulatore**) per calcolare l'informazione cumulativa
  - una **operazione di accumulazione**, binaria, che deve essere applicata all'accumulatore e a ciascun elemento della sequenza
  - inizialmente all'accumulatore viene assegnato l'elemento neutro dell'operazione di accumulazione

Accumulazione: esempi

- Somma della sequenza di n numeri
- Calcolo del fattoriale di un numero

Somma degli elementi di una sequenza

```
int numero;    // elemento corrente della sequenza
    int somma;    // somma degli elementi della sequenza

    /* leggi una sequenza di numeri interi e
     * calcolane la somma */
    /* inizialmente somma vale zero */
    somma = 0;
    /* finché ci sono altri elementi nella sequenza,
     * leggili e sommali a somma */
    while (!Lettore.in.eoln()) {    // finché ci sono
                                    // altri elementi
        /* leggi un elemento della sequenza */
        numero = Lettore.in.leggiInt();
        /* incrementa somma di numero */
        somma = somma + numero;
    }
    /* visualizza somma */
    System.out.println(somma);
```

- sequenza: letta dalla tastiera
- accumulatore: somma
- operazione di accumulazione: +
- elemento neutro dell'operazione di accumulazione: 0

Uso dell'accumulatore

- **Si consideri la variabile somma nel calcolo della somma di una sequenza di numeri**
- **l'obiettivo della variabile somma è quello di memorizzare la somma di tutti gli elementi letti**
  - questo obiettivo viene raggiunto solo dopo aver letto tutti gli elementi della sequenza
- **più precisamente**
  - prima dell'esecuzione dell'istruzione ripetitiva **somma** vale zero – è uguale alla somma di tutti gli elementi letti fino a quel momento
  - dopo ciascuna esecuzione del corpo dell'istruzione ripetitiva, **somma** vale la somma di tutti gli elementi letti fino a quel momento
- **in generale**
  - **somma** è uguale alla somma della porzione di sequenza letta ed elaborata fino a quel momento

Accumulazione: schema risolutivo

```
... dichiarazione della variabile accumulatore ...
  accumulatore = elemento neutro dell'operazione
                 di accumulazione ;
  ... altre inizializzazioni ...
  per ciascun elemento della sequenza {
      ... accedi al prossimo elemento ...
      accumulatore = applica l'operazione di
accumulazione  all'accumulatore e all'elemento corrente
;
  }
  ... altre elaborazioni ...
```

usare un nome opportuno per l'accumulatore

Calcolo di una sottostringa…

Scrivere un metodo di classe String sottostringa(String s, int inizio, int fine) che

- calcola la sottostringa di **s** che consiste dei caratteri compresi tra quello di posizione **inizio** e quello di posizione **fine-1**
  - si comporta come **s.substring(inizio, fine)**
  - ad esempio, **sottostringa("automobile", 2, 6)** deve restituire **"tomo"**
- usa solo le seguenti operazioni sulle stringhe
  - **int length()**
  - **char charAt()**
  - **+** (concatenazione)

…Calcolo di una sottostringa…

È un problema di accumulazione

- sequenza
  - la sequenza dei caratteri di **s** tra le posizioni **inizio** (compresa) e **fine** (esclusa)
- accumulatore
  - una stringa, **ss**
- operazione di accumulazione
  - la concatenazione **+**
- elemento neutro dell'operazione di accumulazione
  - la stringa vuota **""**

...Calcolo di una sottostringa

```
/* Calcola la sottostringa di s compresa tra i
 * caratteri di posizione da inizio a fine-1. */
public static String sottostringa(String s,
                                  int inizio, int fine) {
    // pre: s!=null &&
    //      inizio>=0 && inizio<=s.length() &&
    //      fine>=inizio && fine<=s.length()
    String ss;    // la sottostringa di s tra inizio e fine
    int i;        // indice per la scansione di s

    /* calcola la sottostringa, come concatenazione
     * dei caratteri tra la posizione inizio (inclusa)
     * e fine (esclusa) */
    ss = "";         // la stringa vuota è l'elemento neutro
                     // della concatenazione
    for (i=inizio; i<fine; i++)
        ss = ss + s.charAt(i);  // concatenazione a destra
                                // e conversione automatica
    return ss;
}
```

Esercizio

- **Scrivere un metodo inversa che ha come parametro una stringa S e che calcola la stringa ottenuta da S invertendone l'ordine dei caratteri**
  - ad esempio, **inversa("automobile")** deve restituire **"elibomotua"**

Conteggio

I problemi di conteggio sono un caso particolare dei problemi di accumulazione

- l'accumulatore è un **contatore**
- l'operazione di accumulazione è un **incremento unitario** – vedi oggetto **Contatore**
- l'aggiornamento deve essere eseguito sempre (conteggio), oppure condizionatamente (conteggio condizionale) al soddisfacimento di una proprietà da parte dell'elemento corrente della sequenza
- ad esempio
  - **leggi una sequenza di numeri e calcola la sua lunghezza – conteggio**
  - **leggi una sequenza di numeri e calcola il numero degli elementi che valgono zero – conteggio condizionale**
  - **calcola il numero di occorrenze di un carattere in una stringa – conteggio condizionale**

Conteggio: schema risolutivo

```
int contaProprietà;    // numero di elementi che
                       // soddisfano la proprietà

contaProprietà = 0;
... altre inizializzazioni ...

per ciascun elemento della sequenza {
    ... accedi al prossimo elemento ...
    if (l'elemento corrente soddisfa la proprietà)
        contaProprietà++;
}
... altre elaborazioni ...
```

usare un nome opportuno per il contatore

Esercizio

Scrivere una applicazione che legge dalla tastiera una sequenza di numeri interi e ne conta e visualizza, separatamente, il numero degli elementi positivi e il numero degli elementi negativi (gli zeri non vanno contati)

- Scrivi una sequenza di numeri interi
- **10 20 0 -10 4 -8**
- La sequenza contiene 3 elementi positivi e 2 negativi

Verifica esistenziale

- bisogna determinare se una sequenza di elementi contiene almeno un elemento che soddisfa una certa proprietà

- in altre parole:

si vuole verificare che in una sequenza di elementi $a_1,\ldots, a_n$ esiste almeno un elemento che verifica una data proprietà p cioè che

$$\exists\, i \in \{1,..,n\},\ p(a_i) = \text{true}$$

Verifica esistenziale: schema risolutivo …

- viene usata una variabile booleana che indica se la sequenza contiene almeno un elemento che soddisfa la proprietà
- inizialmente si assegna alla variabile booleana un valore che indica convenzionalmente che la sequenza non contiene nessun elemento che soddisfa la proprietà (false)
- A partire dal primo elemento della sequenza si verifica se l'elemento corrente soddisfa la proprietà
  - se l'elemento corrente soddisfa la proprietà, allora si assegna alla variabile booleana un valore che indica convenzionalmente che la sequenza contiene almeno un elemento che soddisfa la proprietà (true)
- Quando si trova un elemento che soddisfa la proprietà ci si ferma (non ha senso esaminare oltre perché il problema è risolto)

… Verifica esistenziale: schema risolutivo

```
boolean proprietaSoddisfatta;   // almeno un elemento
                                // soddisfa la proprietà

proprietaSoddisfatta = false;
  ... altre inizializzazioni ...

  finche' non trovo un elemento che soddisfa la
   proprieta'
  while (!proprietaSoddisfatta && sequenza non
                                        terminata) {
      ... accedi al prossimo elemento ...
      if (l'elemento corrente soddisfa la proprietà)
          proprietaSoddisfatta = true;
  }
  ... altre elaborazioni ...
```

usare un nome opportuno per la variabile booleana

Un errore comune

Un errore comune nella verifica esistenziale

- ri-assegnare alla variabile booleana usata per la verifica esistenziale il valore che gli è stato assegnato nella sua inizializzazione

```
contieneZero = false;
  while (!Lettore.in.eoln()&& !contieneZero) {
      /* legge il prossimo elemento della sequenza */
      numero = Lettore.in.leggiInt();
      /* se numero vale zero, allora la sequenza
       * contiene almeno uno zero */
      if (numero==0)
          contieneZero = true;
      else
          contieneZero = false;
  }
```

- **Se nella condizione del while non compare `!contieneZero` il programma verifica se l'ultimo elemento della sequenza letta vale zero**
- **Se nella condizione del while compare `!contieneZero` l'ultimo else e' inutile**

Verifica se una sequenza contiene almeno uno zero

```
... legge una sequenza di numeri interi e
      verifica se contiene almeno uno zero ...
  int numero;         // elemento corrente della sequenza
  boolean contieneZero;    // la sequenza contiene almeno
                           // un elemento uguale a zero

  ... visualizza un messaggio ...
  /* legge la sequenza e verifica
   * se contiene almeno uno zero */
  contieneZero = false;
  while (!Lettore.in.eoln()) {
      /* legge il prossimo elemento della sequenza */
      numero = Lettore.in.leggiInt();
      /* se numero vale zero, allora la sequenza
       * contiene almeno uno zero */
      if (numero==0)
          contieneZero = true;
  }
  ... il risultato è contieneZero ...
```

Verifica universale

- **Un problema di verifica universale consiste nel verificare se tutti gli elementi di una sequenza $a_1,\ldots, a_n$ soddisfano una certa proprietà p**
  - una variante (duale) dei problemi di verifica esistenziale
- **In altre parole:**
  - Un problema di verifica universale è soddisfatto, se tutti gli elementi verificano una data proprietà p:

$$\forall\, i \in \{1,..,n\},\ \ p(a_i) = \text{true}$$

  - Oppure un problema di verifica universale non è soddisfatto se esiste almeno un elemento che non verifica p:

$$\exists\, i \in \{1,..,n\},\ \ p(a_i) = \text{false}$$

La verifica universale si può sempre ricondurre alla verifica esistenziale

Verifica universale: schema risolutivo

- **Un problema di verifica universale può essere sempre ricondotto a un problema di verifica esistenziale**
  - il problema diventa quello di verificare se **non** esiste nessun elemento della sequenza che **non** soddisfa la proprietà
  - inizialmente si assegna alla variabile booleana un valore che indica convenzionalmente che tutti gli elementi della sequenza soddisfano la proprietà (**true**)
  - per ogni elemento della sequenza, si verifica se l'elemento corrente **non** soddisfa la proprietà
    - **se l'elemento corrente non soddisfa la proprietà, allora si assegna alla variabile booleana un valore che indica convenzionalmente che non tutti gli elementi della sequenza soddisfano la proprietà (false)**

Verifica universale:schema risolutivo

```
boolean proprietaSoddisfatta;

/* assumo che tutti gli elementi soddisfano
   la proprieta' */
   proprietaSoddisfatta = true;

... altre inizializzazioni ...

finche' non trovo un elemento che non soddisfa
la proprieta
   while (proprietaSoddisfatta &&
                          la sequenza non e' finita){
      ... accedi al prossimo elemento ...
      if (l'elemento corrente non soddisfa la
                                          proprietà)
          proprietàSoddisfatta = false;
}

... altre elaborazioni ...
```

usare un nome opportuno per la variabile booleana

Verifica se una sequenza di dieci elementi è crescente

```
... legge una sequenza di dieci numeri interi e
      verifica se è crescente ...
  int numero;          // elemento corrente della sequenza
  int precedente;      // elemento che precede numero
                       // nella sequenza
  int i;               // per contare i numeri letti
  boolean crescente;   // la sequenza è crescente

  /* il primo elemento della sequenza è il
   * precedente del prossimo che sarà letto */
  precedente = Lettore.in.leggiInt();
  /* finora la sequenza letta è crescente */
  crescente = true;   i=0;
  /* legge e elabora gli altri elementi della sequenza */
  while(i<10 && crescente) {
      /* legge il prossimo numero e verifica che
       * la sequenza sia ancora crescente */
      numero = Lettore.in.leggiInt();
      if (precedente>=numero)
          crescente = false; i++;
      /* prepara la prossima iterazione */
      precedente = numero;
  } ... il risultato della verifica è crescente ...
```

Verifica l'uguaglianza tra stringhe

- **Scrivere un metodo boolean uguali(String s, String t) che verifica se le stringhe s e t sono uguali**
  - si comporta come **s.equals(t)**
    - **ad esempio, uguali("alfa", "alfa") deve restituire true, mentre uguali("alfa", "beta") deve restituire false**
  - due stringhe s e t sono uguali se
    - **s e t hanno la stessa lunghezza**
    - **ciascun carattere della stringa s è uguale al carattere della stringa t che occupa la stessa posizione**

Uguaglianza tra stringhe

Algoritmo:

- **continua il confronto tra le stringhe solo se finora non sono state riscontrate differenze**

```
 /* verifica se s e t sono uguali */
if (s.length()==t.length()) {
    /* s e t hanno la stessa lunghezza: s e t
    /* possono essere uguali, ma sono diverse
     * se contengono almeno un carattere diverso */
    uguali = true;
    i = 0;
    while (uguali && i<s.length()) {
        if (s.charAt(i) != t.charAt(i))
            uguali = false;
        i = i+1;
    }
else /*s e t hanno lunghezza diversa, quindi sono diverse */
   uguali = false;
```

Verifica esistenziale e verifica universale

❑ Verifica esistenziale:

- Si vuole verificare che in una sequenza di elementi

  $a_1,\ldots, a_n$ esiste almeno un elemento che verifica una data proprietà p cioè che

$$\exists\ i \in \{1,..,n\},\ p(a_i) = \text{true}$$

❑ Verifica universale:

- Si vuole verificare che in una sequenza di elementi

  $a_1,\ldots, a_n$ tutti gli elementi della sequenza verificano una data proprietà p cioè che

$$\forall\ i \in \{1,..,n\},\ p(a_i) = \text{true}$$

- Oppure si può verificare che esiste almeno un elemento che non verifica la proprietà p: in tal caso la verifica universale non è soddisfatta

$$\exists\ i \in \{1,..,n\},\ p(a_i) = \text{false}$$

Esercizi

- **Scrivere un metodo boolean primo(int n) che verifica se il numero naturale n è primo**
  - ad esempio, primo(8) deve restituire false
- **Scrivere un metodo boolean caratteriUguali(String s) che verifica se i caratteri di s sono tra loro tutti uguali**
  - ad esempio, caratteriUguali("aaa") deve restituire true
- **Scrivere un metodo boolean prefisso(String s, String t) che verifica se la stringa t è un prefisso della stringa s**
  - ad esempio, prefisso("alfabeto", "alfa") deve restituire true
- **Scrivere un metodo boolean suffisso(String s, String t) che verifica se la stringa t è un suffisso della stringa s**
  - ad esempio, suffisso("aereo", "reo") deve restituire true

Esercizi

- **Scrivere un metodo boolean palindroma(String s) che verifica se la stringa s è palindroma**
  - **ad esempio, palindroma("abcba") deve restituire true**

- **Scrivere un metodo**

  boolean anagramma(String s, String t)

  che verifica se le stringhe s e t sono tra loro degli anagrammi
  - **ad esempio, anagramma("carpa", "capra") deve restituire true**

Ricerca

I problemi di ricerca sono un caso più generale dei problemi di verifica esistenziale

- bisogna verificare se una sequenza contiene almeno un elemento che soddisfa una certa proprietà
- in caso affermativo, bisogna calcolare delle ulteriori informazioni circa uno degli elementi che soddisfa la proprietà
- esempio
  - **ricerca di un numero telefonico in un elenco telefonico a partire dal cognome e nome di un utente**

Indice di un carattere in una stringa

- **Scrivere un metodo**

  int posizioneDi(String s, char car) che

- verifica se la stringa **s** contiene almeno un carattere uguale a **car**
- restituisce la posizione della prima occorrenza del carattere **car** nella stringa **s**
  - oppure il valore **-1** – se **s** non contiene nessun carattere uguale a **car**
- si comporta come **s.indexOf(car)**
  - ad esempio, **posizioneDi("alfa", 'f')** deve restituire **2**, **posizioneDi("alfa", 'a')** deve restituire **1**, mentre **posizioneDi("alfa", 'b')** deve restituire **-1**

Indice di un carattere in una stringa

```
/* Verifica se la stringa s contiene il carattere car,
 * restituisce la posizione della prima
 * occorrenza di car in s, oppure -1. */
public static int posizioneDi(String s, char car) {
    // pre: s!=null
    int posizione;    // posizione di car in s
    int i;          // indice per la scansione di s

    /* scandisce i caratteri di s alla ricerca della
     * prima occorrenza di car in s */
    /* finora non è stata trovato nessun car in s */
    posizione = -1;
    i=0;
    while (posizione==-1 && i<s.length()) {
        if (s.charAt(i)==car)
            /* è stato trovato car in posizione i */
            posizione = i;
        i = i+1;
    }
    return posizione;
}
```

Indice di un carattere in una stringa

```
/* finora non è stata trovato nessun car in s */
posizione = -1;
 i = 0;
 while(posizione==-1 && i<s.length()) {
      if (s.charAt(i)==car)
          /* è stato trovato car in posizione i */
          posizione = i;
      i++;
  }
```

- La variabile **posizione** viene utilizzata come segue
  - **posizione** rappresenta la posizione della prima occorrenza di **car** in **s**
  - **posizione** vale **-1** se **s** non contiene nessuna occorrenza del carattere **car**
    - il valore **-1** denota un "**posizione non valida**" **s**
  - inizialmente **posizione** vale **-1** e, scandendo i caratteri di **s**, se ne viene trovato uno uguale a **car**, la posizione di questo carattere viene assegnata a **posizione**

Ricerca: schema risollutivo

```
... info;       // informazione associata
                // all'elemento cercato

  info = informazione non trovata ;
  ... altre inizializzazioni ...

  finche' non trovo un elemento della sequenza
    che soddisfa la proprieta' {
      ... accedi al prossimo elemento ...
      if (l'elemento corrente soddisfa la proprietà)
          info = informazione associata
                 all'elemento corrente ;
  }
  ... altre elaborazioni ...
```

Indice di una sottostringa in una stringa

Scrivere un metodo

int posizioneDi(String s, String t) che

- verifica se la stringa **s** contiene almeno una sottostringa uguale alla stringa **t**
- restituisce la posizione in cui inizia la prima sottostringa di **s** uguale a **t**
  - **oppure -1 se s non contiene nessuna sottostringa uguale a t**
- **si comporta come s.indexOf(t)**
  - **ad esempio, posizioneDi("sottostringa", "otto") deve restituire 1, posizioneDi("mamma", "ma") deve restituire 0,**

    mentre posizioneDi("mamma", "pa") deve restituire -1

Indice di una sottostringa in una stringa

```
/* Verifica se la stringa s contiene la sottostringa t
 * e restituisce la posizione della prima
 * occorrenza di t in s, oppure -1. */
public static int posizioneDi(String s, String t) {
    // pre: s!=null && t!=null
    int posizione;    // posizione della sottostringa t in s
    int ls, lt;       // lunghezza di s e lunghezza di t
    int i;            // indice per la scansione
                      // delle sottostringhe di s
    int j;            // indice per la scansione di t
    boolean ssug;     // la sottostringa di s che inizia in
                      // i ed ha lunghezza lt è uguale a t

    /* inizializzazioni */
    ls = s.length();
    lt = t.length();

    ... segue ...
```

Indice di una sottostringa in una stringa

```
  /* scandisce le sottostringhe di s
     * alla ricerca di una sottostringa uguale a t */
    posizione = -1;
    for (i=0; posizione==-1 && i<=ls-lt; i++) {
        /* scandisce i caratteri di t per verificare
         * se la sottostringa di s iniziante
         * in posizione i e lunga lt è uguale a t */
        ssug = true;
        for (j=0; ssug && j<lt; j++) {
            if (s.charAt(i+j) != t.charAt(j))
                ssug = false;
            j++;    // passa al prossimo carattere di t
        }
        if (ssug)    // sottostringa di s uguale a t trovata
            posizione = i;   // allora t è sottostringa di s
    }
    return posizione;
}
```

Un’altra soluzione

```
public static int posizioneDi(String s, String t) {
    // pre: s!=null && t!=null
    int posizione;    // posizione della sottostringa t in s
    int ls, lt;       // lunghezza di s e lunghezza di t
    int i;            // indice per la scansione
                      // delle sottostringhe di s

    /* inizializzazioni */
    ls = s.length();
    lt = t.length();

    /* scandisce le sottostringhe di s alla ricerca di
     * una sottostringa uguale a t */
    posizione = -1;
    for (i=0; posizione==-1 && i<=ls-lt; i++) {
        if ( uguali(s,i,i+lt,t) )
            posizione = i;
    }
    return posizione;
}
```

Esercizi

- **Scrivere un metodo int parole(String s) che calcola il numero di parole che compaiono in s**
  - per **parola** si intende una sequenza massimale di caratteri contigui che non contiene nessuno spazio bianco
  - ad esempio, **parole("la mia casa")** deve restituire **3**, mentre **parole(" uno,due.tre ")** deve restituire **1**

- **Scrivere un metodo String tagliaSpazi(String s) che calcola e restituisce i caratteri di s ad eccezione degli eventuali spazi bianchi iniziali e finali**
  - ad esempio, **tagliaSpazi(" a b c ")** deve restituire **"a b c"**