Capitolo 12
Ricorsione

Obiettivi del capitolo

- Comprendere il meccanismo della ricorsione
- Capire la relazione esistente tra ricorsione e iterazione
- Analizzare i problemi che sono molto più semplici da risolvere con la ricorsione che non con l'iterazione
- Imparare a "pensare ricorsivamente"
- Essere in grado di usare metodi ausiliari ricorsivi
- Capire come l'uso della ricorsione si ripercuote sull'efficienza di un algoritmo

Numeri triangolari

- Si calcoli l'area di un triangolo di ampiezza *n*
- Ipotizzando che ciascun [ ] abbia area unitaria
- Questo valore viene a volte chiamato *numero triangolare n-esimo*
- Il terzo numero triangolare è 6
  [ ]
  [ ][ ]
  [ ][ ][ ]

Traccia della classe `Triangle`

```
public class Triangle
{
   public Triangle(int aWidth)
   {
      width = aWidth;
   }
   public int getArea()
   {
      . . .
   }
   private int width;
}
```

Triangolo di ampiezza 1

- Il triangolo consiste di un unico quadrato
- La sua area vale 1
- Aggiungiamo al codice il metodo `getArea` di ampiezza 1

```
public int getArea()
{
   if (width == 1) return 1;
   . . .
}
```


Trattare il caso generale

- Supponiamo di conoscere l'area *smallerArea* del triangolo più piccolo

```
[]
[][]
[][][]
[][][][]
```

- L'area del triangolo più grande può essere calcolata in qusto modo:

```
smallerArea + width
```


Trattare il caso generale

- Calcolare l'area del triangolo più piccolo
  - Costruiamo un triangolo più piccolo e chiediamo la misura dell'area

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```


Completiamo il metodo `getArea`

```
public int getArea()
{
   if (width == 1) return 1;
   Triangle smallerTriangle = new Triangle(width - 1);
   int smallerArea = smallerTriangle.getArea();
   return smallerArea + width;
}
```

Calcolo dell'area di un triangolo di ampiezza 4

- Il metodo `getArea` crea un triangolo più piccolo di ampiezza 3
  - Invoca **`getArea`** su tale triangolo
    - Quel metodo crea un triangolo più piccolo di ampiezza 2
      - Invoca `getArea` su tale triangolo
        - Quel metodo crea un triangolo più piccolo di ampiezza 1
        - Invoca `getArea` su tale triangolo
      - Tale metodo restituisce 1

Calcolo dell'area di un triangolo di ampiezza 4

- Il metodo restituisce `smallerArea + width = 1 + 2 = 3`
  - Il metodo restituisce **`smallerArea + width = 3 + 3 = 6`**
- Il metodo restituisce `smallerArea + width = 6 + 4 = 10`

Ricorsione

- Un calcolo ricorsivo risolve un problema utilizzando la soluzione dello stesso problema con i valori più semplici
- Perché una ricorsione termini, devono esistere casi speciali per i dati di ingresso più semplici
- Completiamo l'esempio del Triangolo,utilizzando width <= 0

```
if (width <= 0)  return 0;
```


Ricorsione

- Esistono due requisiti per essere sicuri che la ricorsione funzioni:
  - Ogni invocazione ricorsiva deve semplificare in qualche modo l'elaborazione
  - Devono esistere casi speciali che gestiscano in modo diretto le elaborazioni più semplici

Altri modi per calcolare I numeri triangolari

- L'area di un triangolo è uguale alla somma:

```
1 + 2 + 3 + . . . + width
```

- Utiliziamo un semplice ciclo:

```
double area = 0;
for (int i = 1; i <= width; i++)
   area = area + i;
```

- Utiliziamo una formula matematica:

```
1 + 2 + . . . + n = n × (n + 1)/2
=> width * (width + 1) / 2
```


File `Triangle.java`

```
01: /**
02:    Una forma triangolare composta di quadrati unitari impilati,
       come questa:
03:    []
04:    [][]
05:    [][][]
06:    . . .
07: */
08: public class Triangle
09: {
10:    /**
11:       Costruisce una forma triangolare
12:       @param aWidth l'ampiezza / e l'altezza) del triangolo
13:    */
14:    public Triangle(int aWidth)
15:    {
16:       width = aWidth;
17:    }
```


File `Triangle.java`

```
18:
19:    /**
20:       Calcola l'area del triangolo
21:       @return the area
22:    */
23:    public int getArea()
24:    {
25:       if (width <= 0) return 0;
26:       if (width == 1) return 1;
27:       Triangle smallerTriangle = new Triangle(width - 1);
28:       int smallerArea = smallerTriangle.getArea();
29:       return smallerArea + width;
30:    }
31:
32:    private int width;
33: }
```


File `TriangleTester.java`

```
01: import java.util.Scanner;
02:
03: public class TriangleTester
04: {
05:    public static void main(String[] args)
06:    {
07:       Scanner in = new Scanner(System.in);
08:       System.out.print("Enter width: ");
09:       int width = in.nextInt();
10:       Triangle t = new Triangle(width);
11:       int area = t.getArea();
12:       System.out.println("Area = " + area);
13:    }
14: }
```

File `triangle.java`

Output

```
Enter width: 10
Area = 55
```


Verifica

1. Perchè nel metodo`getArea` l'enunciato `if (width == 1) return 1;` non è necessaro?
2. Come modifichereste il programma per fare in modo che calcoli ricorsivamente l'area di un quadrato?

Risposte

1. Si supponga di omettere la dichiarazione. Quando calcoliamo l'area di un triangolo avente larghezza 1,computiamo l'area del triangolo di larghezza 0 come 0, e poi aggiungiamo 1, per ottenere l'area corretta.

Risposte

1. Computate ricorsivamente l'area più piccola, poi ritornate `smallerArea + width + width - 1.`

[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]

Naturalmente, sarebbe più semplice computare

$$1+0+2+1+3+2+\cdots+n+n-1=\frac{n(n+1)}{2}+\frac{(n-1)n}{2}=n^2.$$

Permutazioni

- Progettiamo una classe che elenchi tutte le permutazioni di una stringa
- Una permutazione è semplicemente una qualsiasi disposizione delle lettere
- La stringa “eat” ha sei permutazioni
  "eat"
  "eta"
  "aet"
  "tea"
  "tae"

Interfaccia pubblica di PermutationGenerator

```
public class PermutationGenerator
{
   public PermutationGenerator(String aWord) { . . . }
   ArrayList<String> getPermutations() { . . . }
}
```


PermutationGeneratorTester.java

```
01: import java.util.ArrayList;
02:
03: /**
04:    Questo programma utilizza il generatore di permutazioni.
05: */
06: public class PermutationGeneratorTester
07: {
08:    public static void main(String[] args)
09:    {
10:       PermutationGenerator generator
11:             = new PermutationGenerator("eat");
12:       ArrayList<String> permutations
                = generator.getPermutations();
13:       for (String s : permutations)
14:       {
15:          System.out.println(s);
16:       }
```


PermutationGeneratorTester.java

```
17:    }
18: }
19:
```

PermutationGeneratorTester.java

Visualizza

```
eat
eta
aet
ate
tea
tae
```


Generare tutte le permutazioni

- Genereremo tutte le permutazioni che iniziano con la lettera 'e' , poi con la 'a' e infine quelle con la 't'
- Per creare le permutazioni che iniziano con la lettera 'e', abbiamo bisogno di conoscere le permutazini della sottostringa "at"
- Questo è lo stesso problema con un dato di ingresso più semplice
- Usiamo la ricorsione

Generare tutte le permutazioni

- `getPermutations:` ciclo che prende in esame tutte le posizioni all'interno della parola che deve essere permutata
- Per ciascuna posizione, calcoliamo la parola più breve che si ottiene eliminando il carattere i-esimo:

```
String shorterWord = word.substring(0, i)
      + word.substring(i + 1);
```


Generare tutte le permutazioni

- Costruiamo poi un generatore di permutazione che fornisca le permutazioni di tale parola più breve

```
PermutationGenerator shorterPermutationGenerator
      = new PermutationGenerator(shorterWord);
ArrayList<String> shorterWordPermutations
      = shorterPermutationGenerator.getPermutations();
```

Generare tutte le permutazioni

- Infine,aggiungiamo il carattere precedentemente escluso a tutte le permutazionidella parola più breve

```
for (String s : shorterWordPermutations)
{
   result.add(word.charAt(i) + s);
}
```

- Caso speciale: la più semplice stringa possibile è la stringa vuota; permutazione unica, se stessa

PermutationGenerator.java

```
01: import java.util.ArrayList;
02:
03: /**
04:    This class generates permutations of a word.
05: */
06: public class PermutationGenerator
07: {
08:    /**
09:       Constructs a permutation generator.
10:       @param aWord the word to permute
11:    */
12:    public PermutationGenerator(String aWord)
13:    {
14:       word = aWord;
15:    }
16:
```


PermutationGenerator.java

```
17:    /**
18:       Gets all permutations of a given word.
19:    */
20:    public ArrayList<String> getPermutations()
21:    {
22:       ArrayList<String> result = new ArrayList<String>();
23:
24:       // The empty string has a single permutation: itself
25:       if (word.length() == 0)
26:       {
27:          result.add(word);
28:          return result;
29:       }
30:
31:       // Loop through all character positions
32:       for (int i = 0; i < word.length(); i++)
33:       {
```


PermutationGenerator.java

```
34:          // Form a simpler word by removing the ith character
35:          String shorterWord = word.substring(0, i)
36:                + word.substring(i + 1);
37:
38:          // Generate all permutations of the simpler word
39:          PermutationGenerator shorterPermutationGenerator
40:                = new PermutationGenerator(shorterWord);
41:          ArrayList<String> shorterWordPermutations
42:                = shorterPermutationGenerator.getPermutations();
43:
44:          // Add the removed character to the front of
45:          // each permutation of the simpler word,
46:          for (String s : shorterWordPermutations)
47:          {
48:             result.add(word.charAt(i) + s);
49:          }
50:       }
```

PermutationGenerator.java

```
51:       // Return all permutations
52:       return result;
53:    }
54:
55:    private String word;
56: }
```


Verifica

3. Quali sono le pernutazioni della parola di quattro lettere beat?
4. La ricorsione che abbiamo progettato per generare le perutazioni si ferma quando deve elaborare una stringa vuota. Con quale semplice modifica la ricorsione si interromperebbe elaborando una stringa di lunghezza 0 o 1?

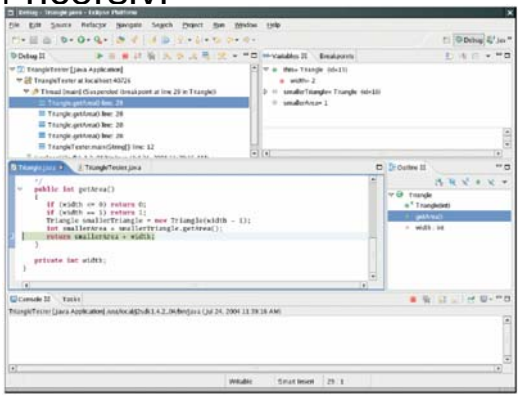
Risposte

3. Sono: `b` seguita dalle sei permutazioni di `eat`, `e` seguita dalle sei permutazioni di `bat`, `a` seguita dalle sei permutazioni di `bet`, e `t` seguita dalle sei permutazioni di `bea`.
4. Basta semplicemente modificare

```
if (word.length() == 0) to if (word.length() <= 1)
```

perchè una parola di una sola lettera è anche l'unica permutazione di se stessa

Tenere traccia di metodi ricorsivi



Pensare ricorsivamente

- Problema: vogliamo verificare se una frase è una palindrome
- Palindrome: una stringa che è uguale a se stessa quando ne invertite l'ordine dei caratteri
  - A man, a plan, a canal–Panama!
  - Go hang a salami, I'm a lasagna hog
  - Madam, I'm Adam

Realizzare il metodo `isPalindrome`

```
public class Sentence
{
   /**
      Constructs a sentence.
      @param aText a string containing all characters of
         the sentence
   */
   public Sentence(String aText)
   {
      text = aText;
   }
```


Realizzare il metodo `isPalindrome`

```
   /**
      Tests whether this sentence is a palindrome.
      @return true if this sentence is a palindrome,
         false otherwise
   */
    public boolean isPalindrome()
   {
      . . .
   }
   private String text;
}
```


Pensare ricorsivamente: Step-by-Step

1. Considerate diversi modi per semplificare i dati. Ecco alcune possibilità:
   - Eliminare il primo carattere
   - Eliminare l'ultimo carattere
   - Eliminare il primo e l'ultimo carattere
   - Eliminare il carattere centrale

Pensare ricorsivamente: Step-by-Step

2. Combinate le soluzioni dei casi più semplici per fornire una soluzione al problema originario.
   - La semplificazione più promettente: eliminare il primo e l'ultimo carattere "adam, I'm Ada"
   - Una parola è una palindroma se
     - La prima e l'ultima lettera sono uguali
     - La parola che si ottiene eliminando la prima e l'ultima lettera è una palindrome

Pensare ricorsivamente: Step-by-Step

2. Combinate le soluzioni dei casi più semplici per fornire una soluzione al problema originario
   - Se il primo e l'ultimo carattere non sono lettere?
     - Se il primo e l'ultimo carattere sono lettere, verificate se sono uguali; In tal caso, eliminateli entrambi e verificate la stringa rimanente.
     - Se l'ultimo carattere non è una lettera, eliminatelo e verificate la stringa rimanente.
     - Se il primo carattere non è una lettera: eliminatelo e verificate la stringa rimanente

Pensare ricorsivamente: Step-by-Step

3. Trovare le soluzioni per i casi più semplici
   - Stringhe di due caratteri
     - Non è necessario identificare una soluzione speciale; la fase 2 si applica anche ad esse
   - Stringhe di un solo carattere
     - Sono palindrome
   - Stringhe vuota
     - Sono palindrome

Pensare ricorsivamente: Step-by-Step

4. Scrivete il codice per la soluzione combinando i casi semplici e il passo di semplificazione

```
public boolean isPalindrome()
{
   int length = text.length();

   // Separate case for shortest strings.
   if (length <= 1) return true;

   // Get first and last characters, converted to lowercase.
   char first = Character.toLowerCase(text.charAt(0));
   char last = Character.toLowerCase(text.charAt(length - 1));
```

Pensare ricorsivamente: Step-by-Step

```
if (Character.isLetter(first) && Character.isLetter(last))
{
    // Both are letters.
    if (first == last)
    {
      // Remove both first and last character.
      Sentence shorter
            = new Sentence(text.substring(1, length - 1));
      return shorter.isPalindrome();
    }
    else
      return false;
    }
```


Pensare ricorsivamente: Step-by-Step

```
   else if (!Character.isLetter(last))
   {
      // Remove last character.
      Sentence shorter
            = new Sentence(text.substring(0, length - 1));
      return shorter.isPalindrome();
    }
    else
    {
      // Remove first character.
      Sentence shorter = new Sentence(text.substring(1));
      return shorter.isPalindrome();
    }
}
```


Metodi ausiliari ricorsivi

- A volte è più semplice trovare una soluzione ricorsiva dopo aver apportato una piccola modifica al problema originario
- Considerate la verifica della palindrome vista nella slide precedente.
  Costruire nuovi oggetti di tipo `Sentence` a ogni passo è poco efficiente.

Metodi ausiliari ricorsivi

- Invece di verificare se l'intera frase è una palindrome, verifichiamo se una sottoriga è una palindrome:

```
/**
   Tests whether a substring of the sentence is a palindrome.
   @param start the index of the first character of the
      substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
public boolean isPalindrome(int start, int end)
```

Metodi ausiliari ricorsivi

- Dopo, invocate semplicemente il metodo ausiliario con valori di posizioni che verifichino l'intera stringa:

```
public boolean isPalindrome()
{
   return isPalindrome(0, text.length() - 1);
}
```


Metodi ausiliari ricorsivi : isPalindrome

```
public boolean isPalindrome(int start, int end)
{
   // Separate case for substrings of length 0 and 1.
   if (start >= end) return true;

   // Get first and last characters, converted to lowercase.
   char first = Character.toLowerCase(text.charAt(start));
   char last = Character.toLowerCase(text.charAt(end));
```


Metodi ausiliari ricorsivi : isPalindrome

```
if (Character.isLetter(first) && Character.isLetter(last))
   {
      if (first == last)
      {
         // Test substring that doesn't contain the
               matching letters.
         return isPalindrome(start + 1, end - 1);
      }
      else
         return false;
   }
   else if (!Character.isLetter(last))
   {
      // Test substring that doesn't contain the last character.
      return isPalindrome(start, end - 1);
   }
```

Metodi ausiliari ricorsivi: isPalindrome

```
if (Character.isLetter(first) && Character.isLetter(last))
   {
      if (first == last)
      {
         // Verifica la sottostringa che non contiene
               le due lettere uguali.
         return isPalindrome(start + 1, end - 1);
      }
      else
         return false;
   }
   else if (!Character.isLetter(last))
   {
      // Verifica la sottostringa che non contiene
            l'ultimo carattere.
        return isPalindrome(start, end - 1);
   }
```

Metodi ausiliari ricorsivi: `isPalindrome`

```
   else
   {
      // Verifica la sottostringa che non contiene
       il primo carattere.
       return isPalindrome(start + 1, end);
   }
}
```


Verifica

5. Dobbiamo necessariamente dare lo stesso nome ad entrambi i metodi `isPalindrome`?
6. In quale momento il metodo ricorsivo `isPalindrome` smette di invocare se stesso?

Risposte

5. No, il primo metodo potrebbe avere un nome diverso,come `substringIsPalindrome`.
6. Quando `start >= end`, cioè quando la stringa esaminata è vuota oppure contiene un solo carattere .

La sequenza di Fibonacci

- La sequenza di Fibonacci è una sequenza di numeri definitada quste equazioni:

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

- I primi dieci valori della sequenza sono,quindi:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

FibTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:    Questo programma calcola i numeri di Fibonacci
05:    usando un metodo ricorsivo.
06: */
07: public class FibTester
08: {
09:    public static void main(String[] args)
10:    {
11:       Scanner in = new Scanner(System.in);
12:       System.out.print("Enter n: ");
13:       int n = in.nextInt();
14:
15:       for (int i = 1; i <= n; i++)
16:       {
```


File FibTester.java

```
17:          long f = fib(i);
18:          System.out.println("fib(" + i + ") = " + f);
19:       }
20:    }
21:
22:    /**
23:       Calcola un numero di Fibonacci.
24:       @param n un numero intero
25:       @return l'n-esimo numero di Fibonacci
26:    */
27:    public static long fib(int n)
28:    {
29:       if (n <= 2) return 1;
30:       else return fib(n - 1) + fib(n - 2);
31:    }
32: }
```

Continua

File PermutationGeneratorTester.jaa

Visualizza

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```


L'efficacia della ricorsione

- L'implementazione ricorsiva del metodo `fib` è semplice e corretta
- Osservate attentamente i dati che vengono visualizzati mentre eseguite il programma
- Le prime invocazioni del metodo `fib` sono abbastanza veloci
- Per valori maggiori,il programma lascia trascorrere una quantità sorprendente di tempo tra due visualizzazioni
- Per identificare il problema inseriamo nel metodo alcuni *messaggi di tracciatura*

FibTrace.java

```
01: import java.util.Scanner;
02:
03: /**
04:    Questo programma stampa messaggi di tracciatura che mostrano
05:    quanto spesso il metodo ricorsivo per calcolare i numeri di
       Fibonacci chiama se stesso.
06: */
07: public class FibTrace
08: {
09:    public static void main(String[] args)
10:    {
11:       Scanner in = new Scanner(System.in);
12:       System.out.print("Enter n: ");
13:       int n = in.nextInt();
14:
15:       long f = fib(n);
16:
17:       System.out.println("fib(" + n + ") = " + f);
18:    }
```


FibTrace.java [2]

```
19:
20:    /**
21:       Calcola un numero di Fibonacci.
22:       @param n un numero intero
23:       @return l'n-esimo numero di Fibonacci
24:    */
25:    public static long fib(int n)
26:    {
27:       System.out.println("Entering fib: n = " + n);
28:       long f;
29:       if (n <= 2) f = 1;
30:       else f = fib(n - 1) + fib(n - 2);
31:       System.out.println("Exiting fib: n = " + n
32:             + " return value = " + f);
33:       return f;
34:    }
35: }
```


FibTrace.java [3]

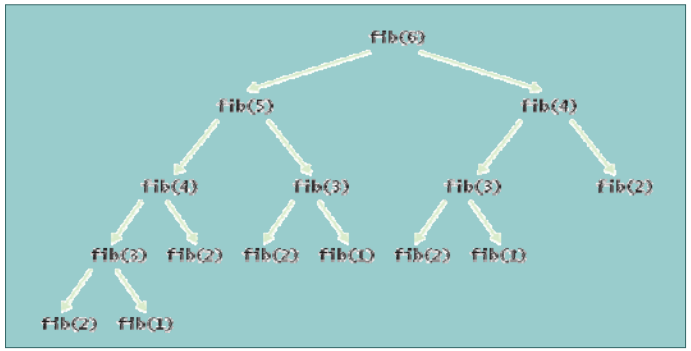
Visualizza:

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
```


FibTrace.java [4]

```
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

Albero delle invocazioni per il calcolo del metodo fib(6)



L'efficacia della ricorsione

- Il metodo è così lento poichè gli stessi valori vengono calcolati più e più volte
- Il calcolo di `fib(6)` invoca `fib(3)` tre volte
- Imitando il procedimento "carta e penna" per permettere di calcolare i valori più di una volta

FibLoop.java

```
01: import java.util.Scanner;
02:
03: /**
04:    Questo programma calcola i numeri di Fibonacci
          usando un metodo interativo.
05: */
06: public class FibLoop
07: {
08:    public static void main(String[] args)
09:    {
10:       Scanner in = new Scanner(System.in);
11:       System.out.print("Enter n: ");
12:       int n = in.nextInt();
13:
14:       for (int i = 1; i <= n; i++)
15:       {
```


FibLoop.java

```
16:          long f = fib(i);
17:          System.out.println("fib(" + i + ") = " + f);
18:       }
19:    }
20:
21:    /**
22:       Calcola un numero di Fibonacci.
23:       @param n un numero intero
24:       @return l'n-esimo numero di Fibonacci
25:    */
26:    public static long fib(int n)
27:    {
28:       if (n <= 2) return 1;
29:       long fold = 1;
30:       long fold2 = 1;
31:       long fnew = 1;
```

FibLoop.java

```
32:       for (int i = 3; i <= n; i++)
33:       {
34:          fnew = fold + fold2;
35:          fold2 = fold;
36:          fold = fnew;
37:       }
38:       return fnew;
39:    }
40: }
```


Permutation GeneratorTester.java

Visualizza:

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```


L'efficacia della ricorsione

- A volte una soluzione ricorsiva viene eseguita molto più lentamente di una soluzione iterativa del medesimo problema
- Nella maggior parte dei casi la soluzione ricorsiva è soltanto poco più lenta
- L'iterativa `isPalindrome` tende ad essere un pochino più veloce di un metodo ricorsivo
  - Ciascuna invocazione di un metodo ricorsivo richiede una certa quantità di tempo di elaborazione del processore

L'efficienza della ricorsione

- Un compilatore efficiente può evitare l'esecuzione di invocazioni ricorsive se queste seguono uno schema semplice.
- La maggior parte dei compilatori non lo fa.
- Solitamente le soluzioni ricorsive sono più facili da capire e da realizzare correttamente, rispetto a soluzioni iterative.
- "Iterare è umano, usare la ricorsione è divino.", L. Peter Deutsch

Il metodo iterativo `isPalindrome`

```
public boolean isPalindrome()
{
   int start = 0;
   int end = text.length() - 1; while (start < end)
   {
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end);
      if (Character.isLetter(first) && Character.isLetter(last))
      {
         // Entrambi sono lettere.
         if (first == last)
         {
            start++;
            end--;
         }
```


Il metodo iterativo `isPalindrome`

```
         else
            return false;
      }
      if (!Character.isLetter(last))
         end--;
      if (!Character.isLetter(first))
         start++;
   }
   return true;
}
```


Verifica

7. La funzione fattoriale può essere calcolata itarativamente, seguendo la definizione $n! = 1 \times 2 \times \ldots \times n$, oppure ricorsivamente, seguendo la definizione secondo cui $0! = 1$ e $n! = (n - 1)! \times n$. In questo caso l'approccio ricorsivo è inefficiente?
8. Perchè non è molto facile sviluppare una soluzione iterativa per il generatore di permutazioni?

Risposte

7. No, la soluzione ricorsiva è all'incirca efficiente quanto la soluzione iterativa:richiedono entrambe l'esecuzione di $n - 1$ moltiplicazioni per calcolare $n!$.

Risposte

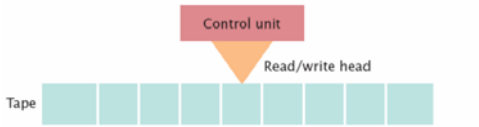
8. Una soluzione iterativa avrebbe un ciclo il cui corpo deve calcolare la permutazione sucessiva a partire dalla permutazione attuale, ma non esiste un modo banale per identificare la permutazione sucessiva. Ad esempio, se avete già trovato la permutazione `eat`, `eta`, e `aet`, non è chiaro come si possa usare questa informazione per trovare la permutazione successiva. In realtà,esiste un ingegnoso algoritmo che risolve questo problema, ma è tutt'altro che banale, come potete vedere nell'Esercizio P12.12.

I limiti del calcolo automatico

I limiti del calcolo automatico

Program

Instruction number	If tape symbol is	Replace with	Then move head	Then go to instruction
1	0	2	right	2
1	1	1	left	4
2	0	0	right	2
2	1	1	right	2
2	2	0	left	3
3	0	0	left	3
3	1	1	left	3
3	2	2	right	1
4	1	1	right	5
4	2	0	left	4



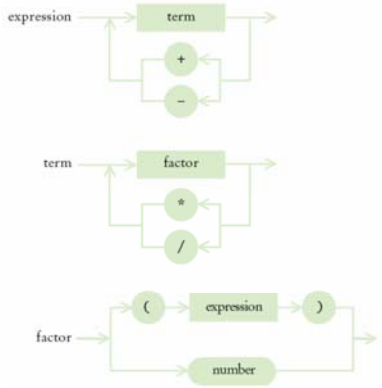
Ricorsione mutua

- Svilupperemo un programma che sia in grado di calcolare i valori di espressioni aritmetiche come:

```
3 + 4 * 5
(3 + 4) * 5
1 - (2 - (3 - (4 - 5)))
```

- Calcolare un'espressione di questo tipo è più complicato perchè:
  - * e / hanno una precedenza più elevata delle operazioni + e -
  - Le parentesi si possono usare per raggruppare sottoespressioni

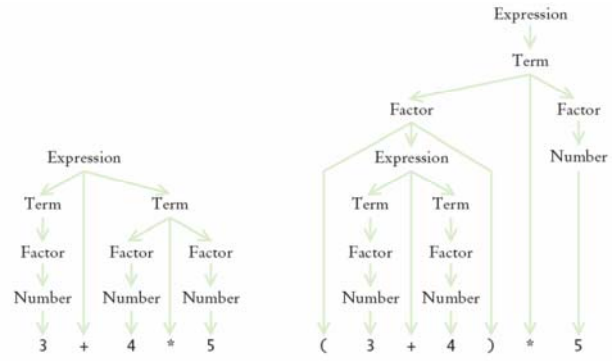
Diagrammi sintattici per la valutazione di un'espressione



Ricorsione mutua

- Un'espressione può essere scomposta in una sequenza di termini, separati da + o -.
- Ciascun termine viene a sua volta scomposto in una sequenza di fattori, separati da * or / .
- Ciascun fattore, infine, è un numero o un'espressione racchiusa fra parentesi tonde.
- Gli alberi sintattici rappresentano l'ordine di esecuzione delle operazioni.

Alberi sintattici per due espressioni.



Metodo di ricorsione mutua

- Nella ricorsione mutua un insieme di metodi cooperanti si invocano l'un l'altro ripetutamente
- Per calcolare il valore di un'espressione realizziamo tre metodi che si invocano l'un l'altro ricorsivamente
  - **`getExpressionValue`**
  - **`getTermValue`**
  - **`getFactorValue`**

Il metodo `getExpressionValue`

```
public int getExpressionValue()
{
   int value = getTermValue();
   boolean done = false;
   while (!done)
   {
      String next = tokenizer.peekToken();
      if ("+".equals(next) || "-".equals(next))
      {
         tokenizer.nextToken(); // Ignora "+" o "-"
         int value2 = getTermValue();
         if ("+".equals(next)) value = value + value2;
         else value = value - value2;
      }
      else done = true;
   }
   return value;
}
```


Il metodo `getFactorValue`

```
public int getFactorValue()
{
   int value;
   String next = tokenizer.peekToken();
   if ("(".equals(next))
   {
      tokenizer.nextToken(); // Ignora "("
      value = getExpressionValue();
      tokenizer.nextToken(); // Ignora ")"
   }
   else
      value = Integer.parseInt(tokenizer.nextToken());
   return value;
}
```


Ricorsione mutua

- Per evidenziare la ricorsione mutua, seguiamo passo passo l'analisi sintattica e la valutazione dell'espressione `(3+4)*5`:
- `getExpressionValue` invoca `getTermValue`
  - **`getTermValue`** invoca **`getFactorValue`**
    - **`getFactorValue`** legge (dalla stringa d'ingresso
    - **`getFactorValue`** invoca **`getExpressionValue`**
      - **`getExpressionValue`** restituisce il valore 7, dopo aver letto **`3 + 4`**; ecco l'invocazione ricorsiva.
    - **`getFactorValue`** legge) dalla stringa di ingresso
    - **`getFactorValue`** restituisce 7

Ricorsione mutua

- **`getTermValue`** legge **`*`** e **`5`** dalla stringa d'ingresso e restituisce **`35`**
- `getExpressionValue` restituisce `35`

Evaluator.java

```
01: /**
02:    Una classe che può calcolare il valore di una espressione.
03: */ aritmetica.
04: public class Evaluator
05: {
06:    /**
07:       Costruisce un valutatore.
08:       @param anExpression una stringa che contiene una
09:       espressione da valutare
10:    */
11:    public Evaluator(String anExpression)
12:    {
13:       tokenizer = new ExpressionTokenizer(anExpression);
14:    }
15:
```


Evaluator.java [2]

```
16:    /**
17:       Valuta l'espressione.
18:       @return il valore dell'espressione.
19:    */
20:    public int getExpressionValue()
21:    {
22:       int value = getTermValue();
23:       boolean done = false;
24:       while (!done)
25:       {
26:          String next = tokenizer.peekToken();
27:          if ("+".equals(next) || "-".equals(next))
28:          {
29:             tokenizer.nextToken(); // Discard "+" or "-"
30:             int value2 = getTermValue();
31:             if ("+".equals(next)) value = value + value2;
32:             else value = value - value2;
33:          }
```

Evaluator.java [3]

```
34:          else done = true;
35:       }
36:       return value;
37:    }
38:
39:    /**
40:       Valuta il sucessivo termine che si trova.
41:       nell'espressione.@return il valore del termine
42:    */
43:    public int getTermValue()
44:    {
45:       int value = getFactorValue();
46:       boolean done = false;
47:       while (!done)
48:       {
49:          String next = tokenizer.peekToken();
50:          if ("*".equals(next) || "/".equals(next))
51:          {
```

Evaluator.java [4]

```
52:             tokenizer.nextToken();
53:             int value2 = getFactorValue();
54:             if ("*".equals(next)) value = value * value2;
55:             else value = value / value2;
56:          }
57:          else done = true;
58:       }
59:       return value;
60:    }
61:
62:    /**
63:       Valuta il successivo termine che si trova
64:          nell'espressione.
       @return il valore del termine
65:    */
```

Evaluator.java [5]

```
66:    public int getFactorValue()
67:    {
68:       int value;
69:       String next = tokenizer.peekToken();
70:       if ("(".equals(next))
71:       {
72:          tokenizer.nextToken(); // Ignora "("
73:          value = getExpressionValue();
74:          tokenizer.nextToken(); // Ignora ")"
75:       }
76:       else
77:          value = Integer.parseInt(tokenizer.nextToken());
78:       return value;
79:    }
80:
81:    private ExpressionTokenizer tokenizer;
82: }
```


Expression Tokenizer.java

```
01: /**
02:    Questa classe scompone una stringa che descrive
03:    un'espressione in elementi:numeri, parentesi tonde,
       e operatori.
04: */
05: public class ExpressionTokenizer
06: {
07:    /**
08:       Costruisce uno scompositore.
09:       @param anInput la stringa da scomporre
10:    */
11:    public ExpressionTokenizer(String anInput)
12:    {
13:       input = anInput;
14:       start = 0;
15:       end = 0;
16:       nextToken();
17:    }
```


Expression Tokenizer.java

```
18:
19:    /**
20:       Restituisce l'elemento successivo senza estrarlo.
21:       @return l'elemento successivo oppure null se non ci
             // sono più elementi
22:    */
23:    public String peekToken()
24:    {
25:       if (start >= input.length()) return null;
26:       else return input.substring(start, end);
27:    }
28:
29:    /**
30:       Restituisce l'elemento successivo e fa avanzare
             // lo scompositore all'elemento successivo.
31:       @return l'elemento successivo o null se non ci sono
             // più elementi
32:    */
```

Expression Tokenizer.java

```
33:    public String nextToken()
34:    {
35:       String r = peekToken();
36:       start = end;
37:       if (start >= input.length()) return r;
38:       if (Character.isDigit(input.charAt(start)))
39:       {
40:          end = start + 1;
41:          while (end < input.length()
42:                && Character.isDigit(input.charAt(end)))
43:             end++;
44:       }
45:       else
46:          end = start + 1;
47:       return r;
48:    }
```

Expression Tokenizer.java

```
50:    private String input;
51:    private int start;
52:    private int end;
53: }
```


Evaluator Tester.java

```
01: import java.util.Scanner;
02:
03: /**
04:    Questo programma calcola il valore di un'espressione
       costituita da numeri,operatori aritmetici e parentesi tonde.
05: */
06: public class EvaluatorTester
07: {
08:    public static void main(String[] args)
09:    {
10:       Scanner in = new Scanner(System.in);
11:       System.out.print("Enter an expression: ");
12:       String input = in.nextLine();
13:       Evaluator e = new Evaluator(input);
14:       int value = e.getExpressionValue();
15:       System.out.println(input + "=" + value);
16:    }
17: }
```


EvaluatorTester.java

- Visualizza:

```
Enter an expression: 3+4*5
3+4*5=23
```


Verifica

9. Qual è la differenza tra termine e fattore? Perchè ci servono entrambi questi concetti?
10. Perchè l'analizzatore sintattico di espressioni("parser")usa la ricorsione mutua?
11. Cosa succede se cercate di analizzare sintatticamente l'espressione `3+4*)5` che non è valida? In particolare,quale metodo lancia un'eccezione?

Risposte

9. I fattori sono tra loro combinati mediante operatori moltiplicativi (`*` e `/`),mentre i termini lo sono mediante operatori additivi (`+`, `-`). Ci servono entrambi questi concetti per fare in modo che la moltiplicazione abbia la precedenza rispetto all'addizione.
10. Per gestire le operazioni con parentesi, come `2+3*(4+5)`. La sotto-espressione `4+5` è gestita da un'invocazione ricorsiva di `getExpressionValue`.

Risposte

11. L'invocazione di `Integer.parseInt` in `getFactorValue` lancia un'eccezione quando trova la stringa `")"`.