Corso di Laurea Ingegneria Informatica
Fondamenti di Informatica 2

Dispensa E02

Esercizi su ricorsione

F. Gasparetti

Febbraio 2008

Contenuti

- Applicazione di test per Stringa palindroma
- Applicazione di test per Hanoi con stampa dell'evoluzione dei perni con "piramide di * "
- Calcolo del numero di attivazioni nel fattoriale
- Applicazione di test per sequenza simmetrica
- Applicazione di test per calcolo dei numeri di Fibonacci
- Definizione ricorsiva della funzione di Ackermann
- Definizioni ricorsive per operazioni algebriche
  - Massimo comun divisore
  - Resto della divisione tra interi
  - Primalità relativa di due numeri
- Calcolo di una sottosequenza di una sequenza data

Stringa palindroma . . .

- Riprendiamo la definizione di stringa palindroma: una stringa che coincide con la stringa stessa letta da destra verso sinistra
- Caratterizzazione induttiva:
  - la stringa vuota è palindroma – p.b.
  - una stringa costituita da un singolo carattere è palindroma – p.b.
  - una stringa *c s d* è palidroma, se *s* è una stringa palidroma e *c* ed *d* sono due caratteri uguali – p.i.
  - nient'altro è una stringa palindroma

. . . Stringa palindroma . . .

- **Problema: Si vuole determinare se una data stringa è palindroma o no.**
- **Dati di ingresso:**
  - **Un oggetto s di tipo String**
- **Pre-condizioni:**
  - **Oggetto s != null**
- **Dati di uscita:**
  - **true se la stringa è palindroma**

. . . Stringa palindroma . . .

- Pseudo-codice dell'algoritmo ricorsivo:

```
boolean palindroma(stringa)
 IF (lunghezza stringa ≤ 1) THEN
   pal ← TRUE
 ELSE
   pal ← TRUE se il carattere iniziale e
                 finale di stringa sono uguali
   pal ← pal AND palindroma(stringa senza
                 il carattere iniziale e finale)
 END IF
 return pal;
```

. . . Stringa palindroma . . .

- Implementazione dell'algoritmo e metodi di test:

```
public class StringaPalindroma {

  public StringaPalindroma() {}

  // Determina se una stringa non nulla
  // in ingresso e' palindroma o no.
  public static boolean palindroma(String s) {
    boolean pal;
    if (s.length() <= 1)
      pal = true;
    else
      pal = (s.charAt(0)==s.charAt(s.length()-1)) &&
                palindroma(s.substring(1,s.length()-1));
    return pal;
  } . . . segue . . .
```

. . . Stringa palindroma . . .

```
public static void testPalindroma() {
  String s = new String("abba"); // test s palindroma
  System.out.println("'"+s+"' palindroma? "+palindroma(s));
  s = new String("abbc");    // s non palindroma
  System.out.println("'"+s+"' palindroma? "+palindroma(s));
  // stringa vuota (nota: non e' la stringa nulla!)
  s = new String("");
  System.out.println("'"+s+"' palindroma? "+palindroma(s));
  s = new String("z");  // stringa di 1 solo carattere
  System.out.println("'"+s+"' palindroma? "+palindroma(s));
}

public static void main(String[] args) {
  testPalindroma();
}
}
```

. . . Stringa palindroma

- Output del programma:

```
'abba' palindroma? true
'abbc' palindroma? false
'' palindroma? true
'z' palindroma? true

C:>
```

Torri di Hanoi . . .

- **Ricordiamo il problema delle Torri di Hanoi: spostare una torre di dischi seguendo le seguenti regole:**
  1. **inizialmente, la torre di dischi di dimensione decrescente è posizionata su un perno 1;**
  2. **l'obiettivo è quello di spostarla su un perno 2, usando un perno 3 di appoggio;**

. . . Torri di Hanoi . . .

3. le condizioni per effettuare gli spostamenti sono:
   1. tutti i dischi, tranne quello spostato, devono stare su una delle torri
   2. è possibile spostare un solo disco alla volta, dalla cima di una torre alla cima di un'altra torre;
   3. un disco non può mai stare su un disco più piccolo

. . . Torri di Hanoi . . .

- Problema: spostare una torre di n dischi dal perno 1 al 2 seguendo le regole suddette
- Dati di ingresso:
  - Numero n di dischi da spostare; perno iniziale e perno destinazione
- Pre-condizioni:
  - $n > 1$
- Post-condizioni:
  - La sequenza di spostamenti rispetta le regole
- Dati di uscita:
  - Sequenza di spostamenti dei singoli dischi

. . . Torri di Hanoi . . .

- Implementazione ricorsiva per il problema delle Torri di Hanoi:

```
public class Hanoi {
 Hanoi() { }

 private static void muoviUnDisco(int sorg, int dest) {
   System.out.println("muovi un disco da "+sorg+" a "+dest);
 }

 private static void muovi(int n, int sorg, int dest, int
   aux){
    if (n == 1)
      muoviUnDisco(sorg, dest);
    else {
      muovi(n-1, sorg, aux, dest);
      muoviUnDisco(sorg, dest);
      muovi(n-1, aux, dest, sorg);
    }
 } . . . segue . . .
```

. . . Torri di Hanoi . . .

```
 . . .
 public static void testHanoi() {
   /* muovere n dischi da 1 a 2
    * con 3 come appoggio */
   int n = 3;
   System.out.println("n = " + n);
   muovi(n, 1, 2, 3);
   n = 1;
   System.out.println("n = " + n);
    muovi(n, 1, 2, 3);
   n = 10;
   System.out.println("n = " + n);
   muovi(n, 1, 2, 3);
 }

 public static void main(String[] args) {
   testHanoi();
 }
}
```

. . . Torri di Hanoi . . .

- Output del programma:

```
n = 3
muovi un disco da 1 a 2
muovi un disco da 1 a 3
muovi un disco da 2 a 3
muovi un disco da 1 a 2
muovi un disco da 3 a 1
muovi un disco da 3 a 2
muovi un disco da 1 a 2
n = 1
muovi un disco da 1 a 2
n = 10
                (seguono 1023 mosse)
```

. . . Torri di Hanoi . . .

- Per esercizio vogliamo stampare lo stato dei perni ad ogni movimento di dischi
- Sfruttiamo l'esercizio *visualizza triangolo di asterischi* fatto nella dispensa 7 "Istruzioni ripetitive" di Fdl-1
- Nota: nel programma precedente lo stato attuale dei dischi è "memorizzato" nella pila delle attivazioni perciò è impossibile accedervi interamente in ogni istante

. . . Torri di Hanoi . . .

```
public class Hanoi {
 int[][] perni;    // memorizza lo stato attuale dei perni
 int num_dischi;

 Hanoi(int n) {
   perni = new int[n][3];
   num_dischi = n;
   for (int i = 0; i < n; i++)
     perni[i][0] = n-i;
   stampa();
 }

 private void muoviUnDisco(int sorg, int dest) {
   System.out.println("muovi un disco da "+sorg+" a "+dest);
   int i, j;
   for (i=num_dischi-1;(i>0) &&(perni[i][sorg-1]==0);i--);
   for (j=num_dischi-1;(j>=0)&&(perni[j][dest-1]==0);j--);
   . . . segue . . .
```

. . . Torri di Hanoi . . .

```
    perni[++j][dest-1] = perni[i][sorg-1];
    perni[i][sorg-1] = 0;
    stampa();
}

public void stampa() {
    for(int i = num_dischi-1; i >= 0; i--) {
      for(int j = 0; j < 3; j++) {
        int l = perni[i][j];
        for(int k = 0; k < num_dischi-l; k++)
          System.out.print(" ");
          for(int k = 0; k < l*2; k++)
            System.out.print("*");  //stampa l*2 asterischi
          for(int k = 0; k < num_dischi-l; k++)
            System.out.print(" ");
          System.out.print("  ");
        }
        System.out.println();
      }   . . . segue . . .
```

. . . Torri di Hanoi . . .

```
    for(int k = 0; k < 4+num_dischi*6; k++)
      System.out.print("-");
    System.out.println();
  }

  private void muovi(int n, int sorg, int dest, int aux) {
      . . . vedi codice precedente . . .
  }

  public static void testHanoi() {
    int n = 3;
    Hanoi hanoi = new Hanoi(n);
    hanoi.muovi(n, 1, 2, 3);
  }

  public static void main(String[] args) {
    testHanoi();
  }
}
```

. . . Torri di Hanoi . . .

❑ Output del programma per n=3:

```
  **
 ****
******
----------------------
muovi un disco da 1 a 2

 ****
******    **
----------------------
muovi un disco da 1 a 3


******    **     ****
----------------------
muovi un disco da 2 a 3

                  **
******            ****
----------------------
(. . . segue . . .)
```

. . . Torri di Hanoi

```
muovi un disco da 1 a 2

                    **
         ******    ****
----------------------
muovi un disco da 3 a 1


  **     ******    ****
----------------------
muovi un disco da 3 a 2

           ****
  **      ******
----------------------
muovi un disco da 1 a 2
            **
           ****
          ******
----------------------


C:>
```

Fattoriale . . .

- Ricordiamo la definizione del fattoriale:

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 \text{ – passo base} \\ n \cdot fatt(n\text{-}1) & \text{se } n > 0 \text{ – passo induttivo} \end{cases}$$

- Vogliamo realizzare e testare un metodo fattoriale per il calcolo ricorsivo che oltre al risultato calcoli anche il numero di attivazioni del metodo

. . . Fattoriale . . .

- Problema: dato un interno n si vuole calcolare ricorsivamente la funzione fattoriale e il numero di attivazioni del metodo per il relativo calcolo
- Dati di ingresso:
  - Un intero n
- Pre-condizioni:
  - $n \geq 0$
- Dati di uscita:
  - Funzione fattoriale di n
  - Numero di attivazioni

. . . Fattoriale . . .

```
public class Fattoriale {
 int natt;          // conteggio numero attivazioni

 public Fattoriale() { }

 public long fattConAtt(long n) {
   natt = 0;
   return _fattConAtt(n);
 }

 public long numeroAttivazioni() {
   return natt;
 }

 private long _fattConAtt(long n) {
   System.out.println("In _fattConAtt("+n+")");
   . . . segue . . .
```

. . . Fattoriale . . .

```
    long f;
    natt++; if (n == 0) {
    System.out.println("Finito");
    f = 1;
  } else {
    System.out.println("Attivazione di
                              _fattConAtt("+(n-1)+")");
    f = n*_fattConAtt(n-1);
    System.out.println("Termine di
                              _fattConAtt("+n+")");
  }
 return f;
}

public static void testFattoriale() {
  Fattoriale fatt = new Fattoriale();
  . . . segue . . .
```

. . . Fattoriale . . .

```
    System.out.println("fattoriale 5 = "+
                                fatt.fattConAtt(5L));
    System.out.println("attivaioni fattoriale 5 = "+
                                fatt.numeroAttivazioni());
    System.out.println("fattoriale 1 = "+
                                fatt.fattConAtt(1L));
    System.out.println("attivazioni fattoriale 1 ="+
                                fatt.numeroAttivazioni());
    System.out.println("fattoriale 0 ="+
                                fatt.fattConAtt(0L));
    System.out.println("attivazioni fattoriale 0 = "+
                                fatt.numeroAttivazioni());
  }

  public static void main(String[] args) {
    testFattoriale();
  }
}
```

. . . Fattoriale . . .

- **Output del programma:**

```
In _fattConAtt(5)
Attivazione di _fattConAtt(4)
In _fattConAtt(4)
Attivazione di _fattConAtt(3)
In _fattConAtt(3)
Attivazione di _fattConAtt(2)
In _fattConAtt(2)
Attivazione di _fattConAtt(1)
In _fattConAtt(1)
Attivazione di _fattConAtt(0)
In _fattConAtt(0)
Finito
(. . . segue . . .)
```

Per il fattoriale di 5 ci sono 6 diverse attivazioni del metodo _fattConAtt

. . . Fattoriale . . .

```
Termine di _fattConAtt(1)
Termine di _fattConAtt(2)
Termine di _fattConAtt(3)
Termine di _fattConAtt(4)
Termine di _fattConAtt(5)
fattoriale 5 = 120
attivaioni fattoriale 5 = 6
In _fattConAtt(1)
Attivazione di _fattConAtt(0)
In _fattConAtt(0)
Finito
Termine di _fattConAtt(1)
fattoriale 1 = 1
attivazioni fattoriale 1 = 2
(. . . segue . . .)
```

. . . Fattoriale

```
In _fattConAtt(0)
Finito
fattoriale 0 = 1
attivazioni fattoriale 0 = 1

C:>
```

Sequenza simmetrica di interi . . .

- Una sequenza di numeri interi tutti positivi tranne che per uno 0 in posizione centrale si dice simmetrica se la sequenza stessa coincide con la sequenza in ordine inverso
- Caratterizzazione ricorsiva:
  - la sequenza costituita solo da 0 è simmetrica (passo base)
  - una sequenza *n s m* è simmetrica, se *s* è simmetrica e *n* ed *m* sono due numeri interi positivi uguali (passo induttivo)
  - nient'altro è una sequenza simmetrica

. . . Sequenza simmetrica di interi . . .

- **Problema: Si vuole determinare se una data sequenza di numeri è simmetrica o no.**
- **Dati di ingresso:**
  - **Un array di interi**
- **Pre-condizioni:**
  - **Dimensione dell'array > 0**
- **Dati di uscita:**
  - **true se la sequenza è simmetrica**

. . . Sequenza di interi simmetrica . . .

```
public class SequenzaInteriSimmetrica {
 public SequenzaInteriSimmetrica() { }

 public static boolean simmetrica(int[] v) {
   return simmetrica(v, 0, v.length-1);
 }

 private static boolean simmetrica(int[] v, int i, int j) {
   boolean sim;
   if (i > j)              sim = false;  // fine array
   else if (v[i] == 0)     sim = true;   // in mezzo all'array
   else {
     boolean b = simmetrica(v, i+1, j-1);
     sim = (v[i] == v[j]) && b;
     return sim;
 } . . . segue . . .
```

. . . Sequenza di interi simmetrica . . .

```
public static void testSimmetrica() {
    int[] v = new int[7];
    v[0] = 1;  v[1] = 2;  v[2] = 3;
    v[3] = 0;
    v[4] = 3;  v[5] = 2;  v[6] = 1;
    System.out.println(simmetrica(v));
    v[0] = 1;  v[1] = 3;  v[2] = 3;
    v[3] = 0;
    v[4] = 3;  v[5] = 2;  v[6] = 1;
    System.out.println(simmetrica(v));
    v = new int[1];
    v[0] = 0;
    System.out.println(simmetrica(v));
    v = new int[3];
    v[0] = 3;
    v[1] = 4;
    System.out.println(simmetrica(v));
}
```

. . . Sequenza di interi simmetrica . . .

```
    public static void main(String[] args) {
        testSimmetrica();
    }
}
```

. . . Sequenza di interi simmetrica . . .

- **Output del programma:**

```
true
false
true
false

C:>
```

Fibonacci . . .

- Ricordiamo la definizione del numero di Fibonacci *n:*

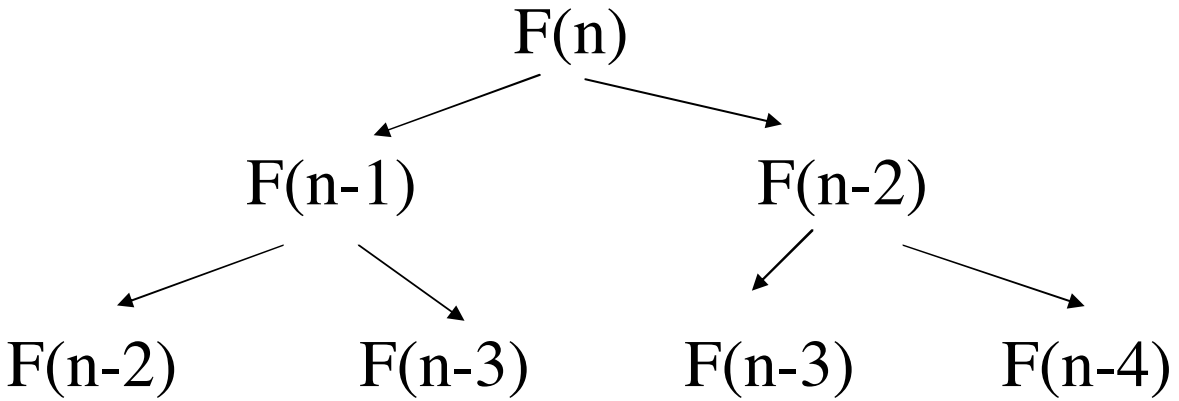
$$F(n) = \begin{cases} 0, & \text{se } n = 0 \text{ – p.b.} \\ 1, & \text{se } n = 1 \text{ – p.b.} \\ F(n-2) + F(n-1) & \text{se } n > 1 \text{ – p.i.} \end{cases}$$

- Sequenza di Fibonacci: *F(0), F(1), F(2), ...*

  cioè: 0, 1, 1, 2, 3, 5, 8, 13, 21

. . . Fibonacci . . .

- **Una implementazione ricorsiva di Fibonacci farà probabilmente uso di ricorsione multipla:**
  - **Il metodo può causare più di una attivazione ricorsiva dello stesso metodo**



. . . Fibonacci . . .

- Problema: dato un interno n si vuole calcolare ricorsivamente la funzione fibonacci e il numero di attivazioni del metodo per il relativo calcolo
- Dati di ingresso:
  - Un intero n
- Pre-condizioni:
  - $n \geq 0$
- Dati di uscita:
  - Fibonacci di n
  - Numero di attivazioni

. . . Fibonacci . . .

- Implementazione ricorsiva per la funzione di Fibonacci:

```
public class Fibonacci {
 int natt;

 Fibonacci() { }

 public  long fibonacci(long n) {
   natt = 0;
   return _fibonacci(n);
 }

 public long numeroAttivazioni() {
   return natt;
 }
 . . . segue . . .
```

. . . Fibonacci . . .

```
private long _fibonacci(long n) {
  System.out.println("In _fibonacci("+n+")");
  natt++;
  long fib;
  if (n == 0) {
    System.out.println("Finito");
    fib = 0;
  } else if (n == 1) {
    System.out.println("Finito");
    fib = 1;
  } else {
    System.out.println("Attivazione di
                        _fibonacci("+(n-1)+")");
    fib = _fibonacci(n-1);

    . . . segue . . .
```

. . . Fibonacci . . .

```
    System.out.println("Attivazione di
                                _fibonacci("+(n-2)+")");
    fib += _fibonacci(n-2);
    System.out.println("Termine di
                                _fibonacci("+n+")");
  }
  return fib;
}

public static void testFibonacci() {
  Fibonacci fib = new Fibonacci();
  System.out.println("fib 8 = "+fib.fibonacci(8L));
  System.out.println("attivazioni fib 8 = "+
                            fib.numeroAttivazioni());
  . . . segue . . .
```

. . . Fibonacci . . .

```
    System.out.println("fib 2 = "+ fib.fibonacci(2L));
    System.out.println("attivazioni fib 2 = "+
                                fib.numeroAttivazioni());
    System.out.println("fib 1 = "+fib.fibonacci(1L));
    System.out.println("attivazioni fib 1 = "+
                                fib.numeroAttivazioni());
    System.out.println("fib 0 = "+fib.fibonacci(0L));
    System.out.println("attivazioni fib 0 = "+
                                fib.numeroAttivazioni());
  }

  public static void main(String[] args) {
    testFibonacci();
  }
}
```

. . . Fibonacci . . .

- Output del programma:

```
In _fibonacci(4)
Attivazione di _fibonacci(3)
In _fibonacci(3)
Attivazione di _fibonacci(2)
In _fibonacci(2)
Attivazione di _fibonacci(1)
In _fibonacci(1)
Finito
Attivazione di _fibonacci(0)
In _fibonacci(0)
Finito
Termine di _fibonacci(2)
(. . . segue . . .)
```

. . . Fibonacci . . .

```
Attivazione di _fibonacci(1)
In _fibonacci(1)
Finito
Termine di _fibonacci(3)
Attivazione di _fibonacci(2)
In _fibonacci(2)
Attivazione di _fibonacci(1)
In _fibonacci(1)
Finito
Attivazione di _fibonacci(0)
In _fibonacci(0)
Finito
Termine di _fibonacci(2)
Termine di _fibonacci(4)
(. . . segue . . .)
```

. . . Fibonacci

```
fib 4 = 3
attivazioni fib 4 = 9
In _fibonacci(1)
Finito
fib 1 = 1
attivazioni fib 1 = 1
In _fibonacci(0)
Finito
fib 0 = 0
attivazioni fib 0 = 1

C:>
```

Funzione di Ackermann . . .

- La funzione di Ackermann è definita come:

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \quad \text{– p.b.} \\ A(m-1,\,1) & \text{se } n = 0 \quad \text{– p.i.} \\ A(m-1,\,A(m,\,n-1)) & \text{altrimenti} \quad \text{– p.i.} \end{cases}$$

- Vogliamo realizzare e testare un metodo ricorsivo per il calcolo della funzione di Ackermann

. . . Funzione di Ackermann . . .

- **Problema: dati due interi *m* e *n*, si vuole calcolare la funzione di Ackermann A(*m*,*n*)**
- **Dati di ingresso:**
  - **Due interi *m* e *n***
- **Pre-condizioni:**
  - ***m* ≥ 0 e *n* ≥ 0**
- **Dati di uscita:**
  - **Un valore intero che corrisponde alla funzione di Ackermann *A(m,n)***

. . . Funzione di Ackermann . . .

- Implementazione ricorsiva per la funzione di Ackermann:

```
public class Ackermann {
 public Ackermann() { }

 public static long ackermann(long m, long n) {
   long ret;
   if (m == 0)
     ret = n+1;
   else if (n == 0)
     ret = ackermann(m-1, 1);
   else
     ret = ackermann(m-1, ackermann(m, n-1));
   return ret;
 }
 . . . segue . . .
```

. . . Funzione di Ackermann . . .

```
 public static void testAckermann() {
  System.out.println("ackermann(0,0)="+ackermann(0L,0L));
  System.out.println("ackermann(2,0)="+ackermann(2L,0L));
  System.out.println("ackermann(0,3)="+ackermann(0L,3L));
  System.out.println("ackermann(3,1)="+ackermann(3L,1L));
  System.out.println("ackermann(3,3)="+ackermann(3L,3L));
 }

 public static void main(String[] args) {
   testAckermann();
 }
}
```

. . . Funzione di Ackermann . . .

- **Output del programma:**

```
ackermann (0,0) = 1
ackermann (2,0) = 3
ackermann (0,3) = 4
ackermann (3,1) = 13
ackermann (3,3) = 61

C:>
```

. . . Funzione di Ackermann

- E' facile dimostrare che la computazione di Ackermann su qualsiasi coppia di interi positivi termina, infatti ad ogni ciclo/chiamata:
  - o si decrementa *m*
  - o se *m* rimane costante, *n* decrementa
  - se *n* va a 0, *m* decrementa

  perciò *m* ed *n* raggiungono sempre 0
- Ma la complessità per valori oltre a 3 è elevata
  - A(x,x) cresce più rapidamente di qualsiasi catena di esponenziali $2^{2^{\cdots^{2^x}}}$

Operazioni albebriche . . .

- Vogliamo implementare metodi ricorsivi sfruttando le seguenti definizioni ricorsive:
- Massimo comun divisore:

$$\text{mcd}(x,y) = \begin{cases} x & \text{se } y = 0 \text{ – p.b.} \\ \text{mcd}(y, r) & \text{se } y > 0 \text{ e } x = q \cdot y + r, \text{ con } 0 \le r < y \text{ – p.i.} \end{cases}$$

- Resto divisione tra un intero e un intero positivo:

$$\text{resto}(x,y) = \begin{cases} \textit{resto}(x + y, y) & \text{se } x < 0 & \text{– p.i.} \\ x & \text{se } 0 \le x < y & \text{– p.b.} \\ \textit{resto}(x - y, y) & \text{se } x > y & \text{– p.i.} \end{cases}$$

. . . Operazioni algebriche . . .

- **Verifica se due numeri interi positivi sono primi tra loro:**

$$primi(x, y) = \begin{cases} true & \text{se } x = 1 \text{ o } y = 1 \\ false & \text{se } x \neq 1, y \neq 1, x = y \\ primi(x, y - x) & \text{se } x \neq 1, y \neq 1, x < y \\ primi(x - y, y) & \text{se } x \neq 1, y \neq 1, x > y \end{cases}$$

. . . Operazioni algebriche . . .

- **Problema (1): determinare il Massimo comun divisore tra due interi**
- **Dati di ingresso:**
  - **Due interi *x* e *y***
- **Pre-condizioni:**
  - ***x* e *y* > 0**
- **Dati di uscita:**
  - **Il numero naturale più grande per il quale possono entrambi essere divisi *x* e *y***

. . . Operazioni algebriche . . .

- **Problema (2): determinare il resto della divisione tra due interi**
- **Dati di ingresso:**
  - Due interi *x* e *y*
- **Pre-condizioni:**
  - $x \geq 0$ e $y > 0$
- **Dati di uscita:**
  - Un intero che denota la quantità da sottrarre a *x* al fine di renderlo divisibile per *y*

. . . Operazioni algebriche . . .

- **Problema (3): verifica se due numeri interi sono primi tra loro**
- **Dati di ingresso:**
  - Due interi *x* e *y*
- **Pre-condizioni:**
  - *x* e *y* > 0
- **Dati di uscita:**
  - false se esiste un numero naturale *n* tale che, *n* > *1* e *x* e *y* siano entrambi divisibili per *n*.

. . . Operazioni algebriche . . .

```
public class Esercizio10_4 {
 public Esercizio10_4() { }

 public static long mcd(long x, long y) {
   long ret = 0L;
   if (y == 0) {
     ret = x;
   } else if (y > 0) {
     long r = x % y;
     /* ((r >= 0) && (r < y)) sempre true */
     ret = mcd(y, r);
   }
   return ret;
 } . . . segue . . .
```

. . . Operazioni algebriche . . .

```
public static boolean primi(long x, long y){
  boolean ret = false;
  if ((x == 1) || (y == 1)){
    ret = true;
  } else if ((x != 1)&&(y != 1)&&(x == y)){
    ret = false;
  } else if ((x != 1)&&(y != 1)&&(x < y)) {
    ret = primi(x, y-x);
  } else if ((x != 1)&&(y != 1)&&(x > y)) {
    ret = primi(x-y, y);
  }
  return ret;
}   . . . segue . . .
```

. . . Operazioni algebriche . . .

```
public static long resto(long x, long y) {
   long ret = 0L;
   if (x < 0) {
      ret = resto(x+y, y);
   } else if (x > y) {
      ret = resto(x-y, y);
   } else if ((x >= 0) && (x < y)) {
      ret = x;
   }
   return ret;
}
. . . segue . . .
```

. . . Operazioni algebriche . . .

```
public static void testMCD() {
  System.out.println("mcd(24,18)="+mcd(24,18));
  System.out.println("mcd(132,48)="+mcd(132,48));
  System.out.println("mcd(7,3)="+mcd(7,3));
  System.out.println("mcd(1,1)="+mcd(1,1));
  System.out.println("mcd(4,1)="+mcd(4,1));
  System.out.println("mcd(1,13)="+mcd(1,13));
}

public static void testPrimi() {
  System.out.println("primi(24,18)="+primi(24,18));
  System.out.println("primi(132,48)="+primi(132,48));
  System.out.println("primi(7,3)="+primi(7,3));
  System.out.println("primi(1,1)="+primi(1,1));
  System.out.println("primi(4,1)="+primi(4,1));
  System.out.println("primi(1,13)="+primi(1,13));
}    . . . segue . . .
```

. . . Operazioni algebriche . . .

```
 public static void testResto() {
   System.out.println("resto(24,18)="+resto(24,18));
   System.out.println("resto(132,48)="+resto(132,48));
   System.out.println("resto(7,3)="+resto(7,3));
   System.out.println("resto(1,1)="+resto(1,1));
   System.out.println("resto(4,1)="+resto(4,1));
   System.out.println("resto(1,7)="+resto(1,7));
 }

 public static void main(String[] args) {
   testMCD();
   testPrimi();
   testResto();
 }
}
```

. . . Operazioni algebriche . . .

- **Output del programma:**

```
mcd(24,18)=6
mcd(132,48)=12
mcd(7,3)=1
mcd(1,1)=1
mcd(4,1)=1
mcd(1,13)=1
primi(24,18)=false
primi(132,48)=false
primi(7,3)=true
primi(1,1)=true
primi(4,1)=true
primi(1,13)=true
(. . . segue . . .)
```

. . . Operazioni algebriche

```
resto(24,18)=6
resto(132,48)=36
resto(7,3)=1
resto(1,1)=0
resto(4,1)=0
resto(1,7)=1


C:>
```

Lunghezza sottostringa . . .

- Fornire l'implementazione di un metodo ricorsivo che, presi come parametri una stringa *s* ed un carattere *c*, restituisca la lunghezza della più lunga sequenza di caratteri *c* consecutivi in *s*.
- Esempio:
  - s = "tprottatttchttttetttx"
  - c = 't'
  - output = 4

. . . Lunghezza sottostringa . . .

- Problema: trovare ricorsivamente in una stringa s la lunghezza della più lunga sottosequenza di caratteri c
- Dati di ingresso:
  - String s e carattere c
- Pre-condizioni:
  - s != null
- Dati di uscita:
  - La lunghezza della sottosequenza più lunga di caratteri c in s

. . . Lunghezza sottostringa . . .

- **Pseudo-codice dell'algoritmo:**

```
int sottoseqRic(stringa, carattere, nc)
IF (stringa vuota) THEN
 len ← nc
ELSE IF (il primo carattere di stringa è c)
 nc ← nc + 1
 len ← sottoseqRic(stringa senza primo carattere, carattere, nc)
 IF len < nc THEN
  len ← nc
 END IF
ELSE
 len ← sottoseqRic(stringa senza primo carattere, carattere, 0)
 IF len < nc THEN
  len ← nc
 END IF
END IF
return len;
```

. . . Lunghezza sottostringa . . .

```
public class Esercizio10_5 {
 public Esercizio10_5() { }

 public static int sottoseqRic(String s, char c) {
   return sottoseqRic(s, c, 0);
 }

 private static int sottoseqRic(String s,char c,int nc) {
   int len;
   if (s.length() == 0) {
     len = nc;
   }
   . . . segue . . .
```

. . . Lunghezza sottostringa . . .

```
  else if (s.charAt(0) == c) {
    nc++;
    len = sottoseqRic(s.substring(1), c, nc);
    if (len < nc) {
      len = nc;
    }
  } else {          // s.charAt(0) != c
    len = sottoseqRic(s.substring(1), c, 0);
    if (len < nc) {
      len = nc;
    }
  }
  return len;
}
. . . segue . . .
```

. . . Lunghezza sottostringa . . .

```
public static void testSottoseqRic() {
  String s = new String("tprottatttchttttetttx");
  char c = 't';
  int max = sottoseqRic(s, c);
  System.out.println("max lung sottoseq  ch "+
                                    c+" in "+s+": "+max);
  s = new String("tt");
  c = 't';
  max = sottoseqRic(s, c);
  System.out.println("max lung sottoseq ch "+
                                    c+" in "+s+": "+max);
  s = new String("");
  c = 't';
  max = sottoseqRic(s, c);
  System.out.println("max lung sottoseq ch "+
                                    c+" in "+s+": "+max);
  . . . segue . . .
```

. . . Lunghezza sottostringa . . .

```
    s = new String("abc");
    c = 't';
    max = sottoseqRic(s, c);
    System.out.println("max lung sottoseq ch"+
                                    c+" in "+s+": "+max);
  }

  public static void main(String[] args) {
    testSottoseqRic();
  }
}
```

. . . Lunghezza sottostringa

- **Output del programma:**

```
max lung sottoseq ch t in
tprottatttchttttetttx: 4
max lung sottoseq ch t in tt: 2
max lung sottoseq ch t in : 0
max lung sottoseq ch t in abc: 0

C:>
```