Programmazione 2

Ricorsione

Prof. Sebastiano Battiato

Prof.ssa Rosalba Giugno

Il calcolo del fattoriale

La funzione fattoriale, molto usata nel calcolo combinatorio, è così definita

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

dove *n* è un numero intero non negativo

Il calcolo del fattoriale

Vediamo di capire cosa significa ....

$$0! = 1$$
$$1! = 1 \cdot (1-1)! = 1 \cdot 0! = 1$$
$$2! = 2 \cdot (2-1)! = 2 \cdot 1! = 2 = 2$$
$$3! = 3 \cdot (3-1)! = 3 \cdot 2! = 3 \cdot 2 \cdot 1 = 6$$
$$4! = 4 \cdot (4-1)! = 4 \cdot 3! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$
$$5! = 5 \cdot (5-1)! = 5 \cdot 4! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

quindi per ogni *n* intero positivo, il fattoriale di *n* è il prodotto dei primi *n* numeri interi positivi

Il calcolo del fattoriale

Scriviamo un metodo statico per calcolare il fattoriale

```
public static int factorial(int n)
{ if(n<0)
      throw new IllegalArgumentException();
  else if (n==0) return 1;
       else
       {    int p=1;
            for (int i=2; i<=n; i++)
                  p=p*i;
            return p;
       }
}
```

Il calcolo del fattoriale

- La funzione così implementata necessita dell' analisi matematica della definizione per scrivere l'algoritmo.
- Realizzando direttamente la definizione, sarebbe stato più naturale scrivere:

```
public static int factorial(int n)
{ if(n<0)
      throw new IllegalArgumentException();
  else if (n==0) return 1;
       else
            return n * factorial(n-1);
}
```

Il calcolo del fattoriale

```
public static int factorial(int n)
{ if(n<0)
      throw new IllegalArgumentException();
  else if (n==0) return 1;
       else
            return n * factorial(n-1);
}
```

E' possibile *invocare un metodo mentre si esegue il metodo stesso*?

Si! E' possibile in java esattamente come in ogni altro linguaggio di programmazione.

La ricorsione

- Invocare un metodo mentre si esegue il metodo stesso è un paradigma di programmazione che si chiama

ricorsione

ed il metodo che ne fa uso si chiama

metodo ricorsivo

- La ricorsione è uno strumento molto potente per realizzare alcuni algoritmi, ma può essere anche causa di molti errori di difficile diagnosi

La pila di esecuzione

Per capire come utilizzare correttamente la ricorsione, vediamo innanzitutto *come funziona*.

In generale, per gestire la chiamata di metodi all'interno del corpo di altri metodi (di cui le funzioni ricorsive sono un caso speciale), la macchina virtuale Java fa uso di una pila di esecuzione (*run-time stack*).

Gli elementi della pila sono record della forma:

Le **var locali** se memorizzate altrove, vengono mantenuti i puntatori alle locazioni dove sono memeorizzate

Parametri e var. locali
Connessione dinamica
Indirizzo di ritorno
Valore restituito

La **connessione dinamica** e' un puntatore alla documentazione di attivazione del chiamante.

e sono detti Documentazione di attivazione (o *activation record*)

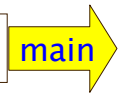
La pila di esecuzione: funzionamento

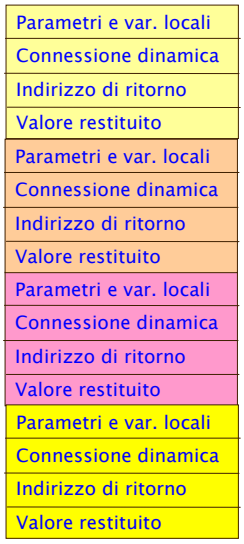
```
main(...){
…
        f1(…);
…}

f1(…){
…
        f2();
…; return;}

f2(…){
…
        f3();
…; return;}
```

Documentazione di attivazione





La ricorsione

Quando un metodo ricorsivo invoca se stesso, *la macchina virtuale Java esegue le stesse azioni che vengono eseguite quando viene invocato un metodo qualsiasi*

1. sospende l' esecuzione del metodo invocante
2. esegue il metodo invocato fino alla sua terminazione
3. riprende l' esecuzione del metodo invocante dal punto in cui era stata sospesa

La ricorsione

Vediamo la sequenza usata per calcolare 3!

si invoca factorial(3)

factorial(3) *invoca* factorial(2)

factorial(2) *invoca* factorial(1)

factorial(1) *invoca* factorial(0)

factorial(0) *restituisce* 1

factorial(1) *restituisce* 1

factorial(2) *restituisce* 2

factorial(3) *restituisce* 6

Si crea quindi una lista di metodi "in attesa", che si allunga e poi si accorcia fino ad estinguersi

La ricorsione

Esistono due regole ben definite che vanno utilizzate per scrivere metodi ricorsivi:

1. individuare il caso base
2. definire il passo ricorsivo

La ricorsione: caso base

Prima regola:

il metodo ricorsivo deve fornire la soluzione del problema in almeno un caso particolare, senza ricorrere ad una chiamata ricorsiva

Tale caso di chiama *caso base*

Nell'esempio del fattoriale il caso base è

```
if (n==0) return 1;
```

Non è necessario che il caso base sia unico

La ricorsione: passo ricorsivo

Seconda regola:

il metodo ricorsivo deve effettuare la chiamata ricorsiva semplificando il problema (ossia avvicinandosi al caso base)

Tale passo di chiama *passo ricorsivo*

Nell'esempio del fattoriale il passo ricorsivo è:

```
if (n>0) return n * factorial(n-1)
```

ed il "problema più semplice" e' calcolare

```
factorial(n-1)
```

La ricorsione: un algoritmo?

Si potrebbe pensare che le chiamate ricorsive si possano succedere una dopo l'altra, all'infinito.

Invece, se

1. ad ogni invocazione il problema diventa sempre più semplice e si avvicina al caso base
2. la soluzione del caso base non richiede ricorsione

allora, per quanto complesso possa essere il problema, certamente la soluzione viene calcolata in un numero finito di passi.

Quindi *la soluzione ricorsiva di un problema è un algoritmo effettivo* in quanto arriva a conclusione in un numero finito di passi

Ricorsione infinita

➢Se manca il caso base, o ad ogni passo ricorsivo la soluzione non si semplifica, il metodo continua a richiamare se stesso all'infinito dando lungo ad una *ricorsione infinita*.

➢Es.

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

➢Solitamente il programma termina con l'errore **StackOverflowError** generato perché si è esaurita la memoria disponibile per tenere traccia delle chiamate.

Ricorsione in coda

- Esistono diversi tipi di ricorsione
- quello visto nel caso del metodo "factorial" si chiama ricorsione in coda (o *tail recursion*)
- Nella ricorsione in coda il metodo ricorsivo esegue una sola invocazione ricorsiva (ossia se stesso), e tale invocazione è *l'ultima azione* del metodo:

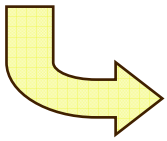
```
public void tail(....)
{ ....
  tail(....);
}
```

Schema

Eliminazione della ricorsione in coda

La ricorsione in coda può essere *eliminata* trasformando il metodo ricorsivo in un *metodo non ricorsivo che fa uso di un ciclo*

```
public static int factorial(int n)
{     if (n==0) return 1;
      else return n * factorial(n-1);}
```



```
public static int factorial(int n)
{     int f=1;
      while (n>0)
      {     f=f*n;
            n--;}
      return f;}
```

Ricorsione in coda

Allora, *a cosa serve la ricorsione in coda?*

- Non è necessaria ma rende il codice più leggibile
- E' utile quando la soluzione del problema è esplicitamente ricorsiva (es. fattoriale)
- In ogni caso, la ricorsione in coda è meno efficiente del ciclo equivalente perché il sistema deve gestire le invocazioni sospese

Esercizio

Scrivere in Java un metodo ricorsivo per il calcolo della funzione esponenziale definita come segue:

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ x \cdot x^{y-1} & \text{se } y > 0 \end{cases}$$

Scrivere in Java la versione non ricorsiva della stessa funzione

Soluzione: versione ricorsiva

caso base: se y=0, esp(x,y)=1

passo ricorsivo: se y>0, esp(x,y)= x * esp(x,y-1)

```
public double esp(double x, int y)
{ if(y<0)
      throw new IllegalArgumentException();
  else if (y==0) return 1.0;
       else
            return x * esp(x,y-1);
}
```

Soluzione: legare “esp” con “main”

```
import java.io.*;
class Esponenziale
{ public static void main(String[] args)
  { ConsoleReader console = new ConsoleReader();
    System.out.print("base = ");
    double base = console.readDouble();
    System.out.print("esponente = ");
    int espon = console.readInt();
    System.out.println(base + " elevato “ + espon + “ = “ +
                       esp(base,espon));
    return;
  }

   public static double esp(double x, int y)
   { if(y<0) throw new IllegalArgumentException();
     else if (y==0) return 1.0;
          else return x * esp(x,y-1);
   }
}
```

Soluzione: versione non ricorsiva

```
public double iterEsp(double x, int y)
{ if(y<0)
      throw new IllegalArgumentException();
  else
  { if(y==0) return 1.0;
    else
    { double ris=x;
      while (y>1)
      { ris=ris*x;
        y--;
      }
      return ris;
    }
  }
}
```

Completare con la funzione **`main`**

Ricorsione non in coda

- E' relativa a metodi ricorsivi in cui *la chiamata al metodo stesso non è l'ultima azione* compiuta
- La ricorsione non in coda non può essere eliminata facilmente (ossia è complesso scrivere un codice equivalente)
- Però è possibile dimostrare che essa è sempre eliminabile

Ricorsione non in coda: un esempio

Un esempio di metodo ricorsivo non in coda è dato dalla seguente funzione che inverte una stringa di input:

```
public void reverse()
{  char ch = (char)System.in.read();
   if (ch != '\n')
   {  reverse();
      System.out.print(ch);
   }
}
```

Ricorsione non in coda: un esempio

Legando reverse con main

```
import java.io.*;
class Reverse
{ public static void main(String[] args)
  { reverse();   }

  static void reverse()
  { char ch;
    try
    { ch = (char) System.in.read();    }
    catch (IOException e)
    { return;                          }
    if (ch != '\n')
    { reverse();
       System.out.print(ch);
    }
  }
}
```

Ricorsione non in coda: un esempio

```
public void reverse()
{  char ch = (char)System.in.read();
   if (ch != '\n')
   {  reverse();
      System.out.print(ch);
   }
}
```

In questo esempio la ricorsione puo' essere eliminata inserendo i dati letti in uno stack.

Implementate la versione iterativa di questo metodo.

Ricorsione multipla

- Si parla di ricorsione multipla quando un *metodo richiama se stesso più volte* durante la sua esecuzione
- Un esempio di ricorsione multipla è la funzione di Fibonacci

$$Fib(n) = \begin{cases} n & \text{se } 0 \le n < 2 \\ Fib(n-2) + Fib(n-1) & \text{se } n \ge 2 \end{cases}$$

E' una funzione che genera i numeri di Fibonacci:

0,1,1,2,3,5,8,13,21,34,55,89,....

i primi due numeri sono 0 ed 1, gli altri sono la somma dei due predecessori nella sequenza prodotta

Funzione di Fibonacci

$$Fib(n) = \begin{cases} n & \text{se } 0 \le n < 2 \\ Fib(n-2) + Fib(n-1) & \text{se } n \ge 2 \end{cases}$$

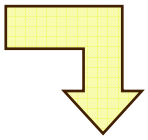
```
public static int fibonacci(int n)
{ if(n<0)
      throw new IllegalArgumentException();
  else if (n<2) return n;
       else
            return fibonacci(n-2) + fibonacci(n-1);
}
```

Ricorsione multipla

- La ricorsione multipla va usata con estrema cautela perché può portare a programmi molto inefficienti
- Eseguendo il calcolo della funzione di Fibonacci per un intero relativamente grande si può osservare che il tempo di elaborazione cresce MOLTO rapidamente
  - ☹ si può calcolare che occorrono 3 milioni di invocazioni per calcolare fibonacci(31) !!!
- A causa della crescita eccessiva del tempo di calcolo, un tipo di ricorsione come quella della funzione di Fibonacci è nota anche come ricorsione eccessiva

Eliminazione della ricorsione multipla

```
public static int fib(int n)
{ if (n<2) return n;
  else return fib(n-2)+fib(n-1);
}
```



la ricorsione multipla è ancora più difficile da eliminare rispetto alla ricorsione semplice, ma è sempre possibile eliminarla

```
public static int iterativefib(int n)
{ if (n<2) return n;
  else
  { int i=2, tmp, current=1, last=0;
    for (;i<=n; ++i)
    {  tmp = current;
       current += last;
       last = tmp;
    }
    return current;
  }
}
```

Torri di Hanoi

Si hanno a disposizione 3 pile di dischi (*torri*) allineate:

- all'inizio tutti i dischi si trovano sulla pila di sx

- alla fine tutti i dischi si devono trovare sulla pila di dx

I dischi sono tutti di dimensione diversa e quando si trovano su una pila devono rispettare la seguente regola

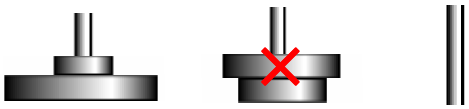
nessun disco può avere sopra di sé dischi più grandi

La leggenda: il mondo finira' quando dei monaci di un certo tempio risolveranno questa problema con 64 dischi d'oro e 3 piloni di diamante.

Torri di Hanoi: regole

1. si può spostare un disco alla volta
2. il disco da spostare deve essere rimosso dalla cima della torre ed inserito in cima ad un'altra torre

   non può essere parcheggiato all'esterno
3. in ogni momento deve essere sempre soddisfatta la condizione che *nessun disco può avere sopra di sé dischi più grandi*

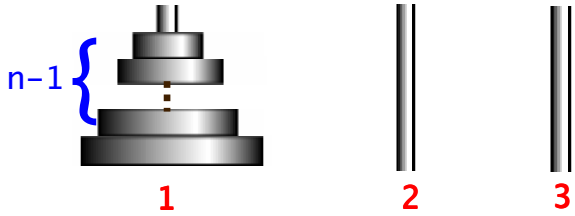


Algoritmo di soluzione

- Per la soluzione del gioco di Hanoi è noto un algoritmo ricorsivo
- Il problema generale consiste nello spostare *n* dischi da una torre all'altra, usando la terza torre come deposito temporaneo
- Per spostare *n* dischi da una torre all'altra si suppone di saper spostare *n* –1 dischi da una torre all'altra
- Per descrivere l'algoritmo identifichiamo le torri con i numeri interi 1,2,3

Algoritmo ricorsivo

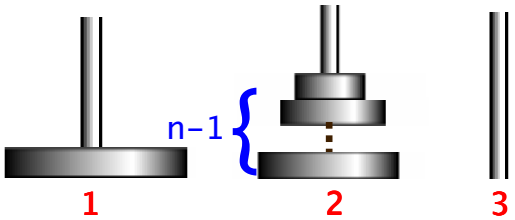
Per spostare *n* dischi dalla torre 1 alla torre 3:



Algoritmo ricorsivo

Per spostare *n* dischi dalla torre 1 alla torre 3:

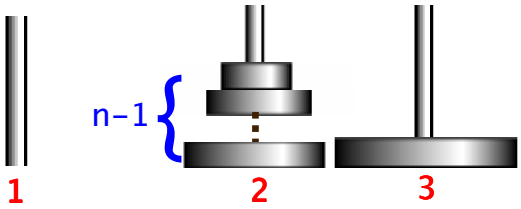
1. gli *n* -1 dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)



Algoritmo ricorsivo

Per spostare *n* dischi dalla torre 1 alla torre 3:

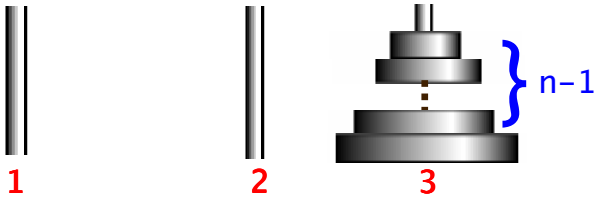
1. gli *n* –1 dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)
2. il disco rimasto sulla torre 1 viene spostato nella torre 3



Algoritmo ricorsivo

Per spostare *n* dischi dalla torre 1 alla torre 3:

1. gli *n* –1 dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)
2. il disco rimasto sulla torre 1 viene spostato nella torre 3
3. gli *n* –1 dischi in cima alla torre 2 vengono spostati sulla torre 3, usando la torre 1 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 1 rimane vuota)



Algoritmo ricorsivo

Per spostare *n* dischi dalla torre 1 alla torre 3:

1. gli *n* –1 dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)
2. il disco rimasto sulla torre 1 viene spostato nella torre 3
3. gli *n* –1 dischi in cima alla torre 2 vengono spostati sulla torre 3, usando la torre 1 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 1 rimane vuota)

Il caso base si ha quando $n = 1$ e l'algoritmo usa il solo passo 2, che non ha chiamate ricorsive

Si dimostra che il numero di mosse per risolvere il gioco con questo algoritmo è $\mathbf{2^n - 1}$

Soluzione delle Torri di Hanoi

```
public class Hanoi
{  public static void solveHanoi (int from, int to,
                                  int temp, int dim)
   { if (dim > 0)
     {  solveHanoi(from,temp,to,dim-1);
        System.out.println("Sposta da "+ from + " a " + to);
        solveHanoi(temp,to,from,dim-1);
     }
  }
  public static void main (String args[])
  { solveHanoi(1,3,2,Integer.parseInt(args[0]));
  }
}
```

Soluzione delle Torri di Hanoi

Output per $n = 4$:

#mosse: $2^4-1= 15$

```
Sposta da 1 a 2
Sposta da 1 a 3
Sposta da 2 a 3
Sposta da 1 a 2
Sposta da 3 a 1
Sposta da 3 a 2
Sposta da 1 a 2
Sposta da 1 a 3
Sposta da 2 a 3
Sposta da 2 a 1
Sposta da 3 a 1
Sposta da 2 a 3
Sposta da 1 a 2
Sposta da 1 a 3
Sposta da 2 a 3
```

Programmazione 2

Esercitazione Ricorsione

Prof. Sebastiano Battiato

Prof.ssa Rosalba Giugno

Esercizi

1. Scrivere un metodo che determini se una stringa in ingresso è palindroma o no, ovvero se il primo carattere e' uguale all'ultimo, il secondo carattere e' uguale al penultimo e cosi' via.

2. Scrivere un metodo per convertire un numero in base 10 nel corrispondente in base 2.

Esercizio 1

```
public static boolean isP(String s)
  {
    if (s.length()==0 || s.length()== 1)
      return true;

    if (isP(s.substring(1,s.length()- 1))
        && s.charAt(0) == s.charAt(s.length()-1) )
      return true;
    return false;

  }
```

Esercizio 2

```
public static String recDecToBin(int n)
  {
    if (n==0) return "";
    return
      recDecToBin(n/2)+(new Integer((n%2))).toString();
  }
```

Attenzione all'ordine dei resti.!!!!

Coefficienti binomiali

Esercizio 5. I coefficienti binomiali sono il risultato dello sviluppo di un'espressione binomiale del tipo $(x+1)^n$. Per esempio:

$$(x+1)^6 = x^6+6x^5+15x^4+20x^3+15x^2+6x+1$$

I 7 coefficienti generati sono in questo caso: 1,6,15,20,15,6,1.

Il matematico Pascal scoprì una relazione ***ricorsiva*** tra i coefficienti binomiali: disponendoli su un triangolo, trovò che ogni numero al suo interno è pari alla somma dei due posti sopra di esso

Coefficienti binomiali

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
```

Si indichi con ***c(n,k)*** il coefficiente che si trova alla riga ***n*** e alla colonna ***k*** (conteggiate da 0).

Per esempio *c(6,2)=15*

La relazione di Pascal assume allora la forma:

c(n,k)=c(n-1, k-1)+c(n-1,k) ***per 0<k<n***

Calcolo di c(n,k) ricorsivo

```
public static int C_n_k(int n, int k) {
   if (k==0 || k==n)
            return 1;
        return C_n_k(n-1,k-1)+C_n_k(n-1,k);
    }
```

Il caso base della ricorsione copre la parte **sinistra** e **destra** del triangolo. Vado cioè a ritroso nell'esplorazione del triangolo di Pascal, fino a quando non raggiungo i "bordi".

Calcolo di c(n,k) iterativo

```
public static int C_n_k_iter(int n, int k) {
        if (n<2 || k==0 || k==n)
            return 1;
        int c=1;
        for (int j=1; j<=k; j++)
            c=c*(n-j+1)/j;
        return c;
```

Si utilizza la formula per il calcolo esplicito dei coefficienti binomiali:

$$c(n,k) = \frac{n!}{k!(n-k)!} = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdots \frac{n-k+1}{k}$$

Algoritmo di Euclide

Esercizio 6. L'algoritmo di Euclide calcola il **massimo comune divisore** di 2 numeri interi positivi. Si tratta probabilmente del più antico algoritmo ricorsivo (circa 300 A.C.).

Dati due numeri *n* ed *m*, con *n*<*m*, si procede sottraendo ripetutamente il numero più piccolo *n* dal più grande *m* fino a quando la differenza *d* tra i due numeri è minore di *n*.

A questo punto si ripete il procedimento con *d* al posto di *n* e con *n* al posto di *m,* fino a quando la differenza *d* non eguaglia esattamente *n.*

Tale valore è il ***massimo comune divisore*** dei due numeri orginali.

Algoritmo di Euclide

Esempio
n=130
m=494
m n d
494-130=364
364-130=234
234-130=104 -> **m n d**
130-104=26 -> **m n d**
104-26= 78
78-26= 52
52-26= 26 (M.C.D)

Versione ricorsiva

```
public static int MCD(int m,int n) {
            if (n==m)
                return n;
            else
                if (n<m)
                    return MCD(m-n,n);
                else
                    return MCD(m,n-m);
          }
```

In certi casi l'algoritmo puo' risultare molto lento; Si puo' usare una versione dell'algoritmo basata sulle divisioni con resto successive

Versione ricorsiva

```
public static int MCD(int m,int n) {
            if (n==0)
                return m;
            return MCD(n,m%n);
}
```

MCD di x,y con x>y e' lo stesso del MCD di y e x mod y.

Un numero t divide sia x che y se e solo se t divide sia y che x mod y, perche' x e' dato dalla somma di x mod y e un multiplo di y.

La funzione di Ackermann

E' un esempio di ricorsione annidata

$$A(n,m)=\begin{cases} m+1 & \text{se } n=0 \\ A(n-1,1), & \text{se } n>0,\ m=0; \\ A(n-1,A(n,m-1)), & \text{altrimenti} \end{cases}$$

Esercizio 7. La funzione di Ackermann *A(m,n)* è l'esempio più semplice di una funzione totale calcolabile ma non primitiva ricorsiva. Questo significa, in pratica, che la funzione di Ackermann cresce **MOLTO** rapidamente. (Esempio $A(3,m)=2^{m+3}-3$; *A(4,m)=2 (elevato ad una pila di m due elevato a 16) -3;* $A(4,1)=2^{65536}-3$*—che e' maggiore del numero di atomi dell'universo*)

La funzione di Ackermann

Si consideri ad esempio che la funzione f(x) = A(x, x) cresce **MOLTO** più rapidamente di qualsiasi polinomio o esponenziale.

```
public static int Ack(int n,int m){
        if (n==0) return (m+1);
        if ((n>0)&&(m==0))
            return Ack(n-1,1);
        else
            return Ack(n-1,Ack(n,m-1));
    }
```

Esprimerla in forma non ricorsiva e' veramente complesso

Seno, Coseno e Ricorsione Mutua

E' un esempio di ricorsione indiretta: f chiama g e g chiama f

Esercizio 8 Calcolare il valore delle funzioni trigonometriche **seno** e **coseno** utilizzando le identità:

$$\sin 2\theta = 2 \sin\theta \cos\theta$$

$$\cos 2\theta = 1 - 2(\sin\theta)^2$$

e lo sviluppo polinomiale di Taylor

$$\sin x = x - x^3/6$$

$$\cos x = 1 - x^2/2$$

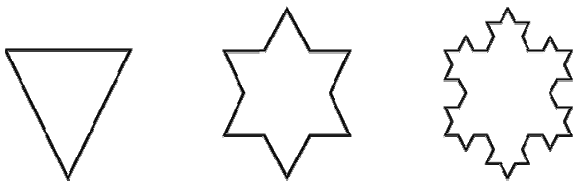
che sono buone approssimazioni delle funzioni per **piccoli** valori di x.

Seno, Coseno e Ricorsione Mutua

```
public static double s(double x)  {
      if (-0.005 < x && x < 0.005)
            return x - x*x*x/6;
      return 2*s(x/2)*c(x/2);
  }

public static double c(double x){
      if (-0.005 < x && x < 0.005)
            return 1.0 - x*x/2;
      return 1 - 2*s(x/2)*s(x/2);
  }
```

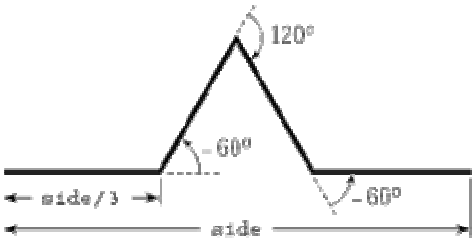
Van Koch e i fiocchi di neve



La curva in questione fu ideata nel 1904 dal matematico svedese Helge von Koch; la costruzione della curva stessa può essere descritta mediante un procedimento ricorsivo.

La curva può essere descritta come una combinazione di tre curve identiche disegnate con angolazioni diverse e unite insieme.

Van Koch e i fiocchi di neve



Una di tali curve può essere così disegnata:

1 Dividi un intervallo *side* in tre parti uguali.

2 Sposta un terzo di *side* nella direzione indicata da *angle.*

3 Gira di -60° e vai avanti per un terzo di *side.*

4 Gira di 120° e vai avanti per un terzo di *side.*

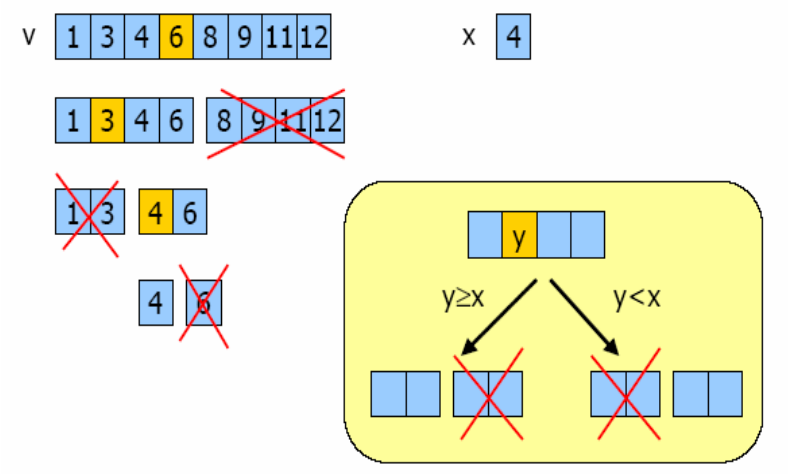
5 Gira -60° e disegna di nuovo una linea lunga un terzo di side.

Ricerca Dicotomica

Esercizio 9. Determinare se un elemento x è presente all'interno di un vettore ordinato v[N];

IDEA: Dividere il vettore a metà e riapplicare il problema su una delle due metà, (l'altra si può escludere a priori, essendo il vettore ordinato)

Ricerca Dicotomica



Ricerca Dicotomica

```
public static int trova(int v[], int a, int b, int x)
{
	int c ;
	if(b-a == 0)
		if(v[a]==x)
			return a ;
		else
			return v.length ;
	c = (a+b) / 2 ;
	if(v[c] >= x)
		return trova(v, a, c, x) ;
	else
		return trova(v, c+1, b, x) ;
}
```

Ricorsione e liste

Strutture dati costruite a partire da oggetti i cui campi sono riferimenti ad altri oggetti dello stesso tipo sono inerentemente ricorsive.

Le liste concatenate si possono definire in modo **ricorsivo**, per cui una funzione o proprietà da testare su una lista, si può calcolare, basandosi appunto su questa definizione **ricorsiva**. Anziché considerare una lista come una sequenza di dati dello stesso tipo (per esempio, interi), utilizziamo le seguente definizione alternativa:

- **una lista è la lista vuota, oppure un elemento seguito da un'altra lista.**

In altre parole, una sequenza di elementi può essere composta da zero elementi, oppure da un elemento seguito da altri elementi in sequenza.

Ricerca di un elemento

```
public static boolean find_rec_LL(Object x, ListNode
  n){

   if (n==null)
      return false;
   else
       if (((Comparable) n.getInfo()).compareTo(x)==0)
                return true;
            else
                return find_rec_LL(x, n.getNext());

    }
```

Stampa di una lista in ordine inverso

```
public static void invert_rec_LL(ListNode n){

        if (n==null)
            return;
        else
           invert_rec_LL(n.getNext());
        System.out.print(n.getInfo()+" ");
       }
```

Inversione di una lista

```
public static LinkedList reverse(LinkedList l)
  {
   if (!l.isEmpty())
    {
     Object n = l.deleteHead();
     reverse(l).insertTail(n);
    }
   return l;
  }
```

Esercizio proposto

Esercizio 13. Si esegua il calcolo del determinante di una matrice quadrata. Si ricorda che:

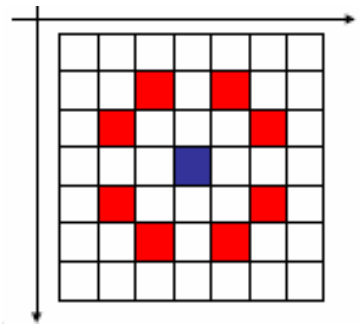
$Det(M_{1x1}) = m_{11}$

$Det(M_{NxN})$ = somma dei prodotti degli elementi di una riga (o colonna) per i determinanti delle sottomatrici $(N-1)x(N-1)$ ottenute cancellando la riga e la colonna che contengono l'elemento, preso con segno $(-1)^{i+j}$.

Il Tour del Cavaliere

Trovare una sequenza di mosse del cavallo tale per cui questo tocca una ed una sola volta ciascuna casella di una scacchiera *NxN*.

Si ricorda che un cavallo in posizione *(i, j)* può muovere in 8 possibili caselle:



Il Tour del Cavaliere

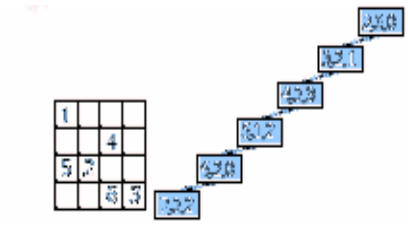
Il numero di mosse possibili ad ogni passo è al più 8.

Il numero di caselle da riempire e quindi il numero di passi è N^2

Quindi l'albero di "decisione" ha un numero di nodi $<= 8N^2$

Il numero di chiamate ricorsive è: $\Theta(8N^2)$.

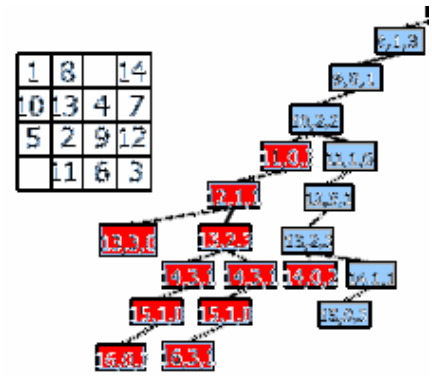
Albero di decisione



Fino a quando trovo caselle libere….procedo…. Tenendo traccia del cammino seguito.

Se trovo una casella occupata.. (11 in (1,1) non può andare in (3,0)) scelgo una possibile alternativa…..

Albero di decisione



- Se non ci sono alternative devo tornare indietro e scegliere ove possibile soluzioni alternative (backtracking)

- Devo ricordarmi la storia …..

Esercizio 12

```
public static void main(String Args[])
   {
        int DIM=8;
        int scacc[][] = new int[DIM][DIM];
        int i, j, result;

        for( i=0; i<DIM; i++)
             for( j=0; j<DIM; j++)
                  scacc[i][j] = 0;
        scacc[0][0] = 1;
        result = muovi(2,0,0,scacc);

        if (result==0)
        System.out.println("Soluzione non trovata\n");
   }
```

Esercizio 12

```
public static int muovi(int mossa, int posx, int
  posy,int[][] scacc){
  int i,j,ret,newposx,newposy,DIM=scacc.length;
  int a[]=new int[8], b[]= new int[8];
  a[0]=2;   b[0]=1;     a[1]=1; b[1]=2;
  a[2]=-1;  b[2]=2;     a[3]=-2;b[3]=1;
  a[4]=-2;  b[4]=-1;    a[5]=-1;b[5]=-2;
  a[6]=1;   b[6]=-2;    a[7]=2; b[7]=-1;

  if( mossa == (DIM*DIM+1)){
      for( i=0; i<DIM; i++){
         for( j=0; j<DIM; j++)
            System.out.println("Posizione:+i+","+j+“
  ..mossanumero:"+scacc[i][j]);
               }
  return 1;}
```

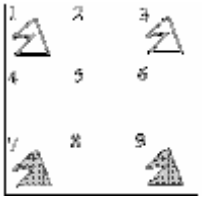
Esercizio 12

```
…
 for( i=0; i<8; i++){
    newposx = posx + a[i];
    newposy = posy + b[i];
    if((newposx<DIM)&&(newposx>=0)
                  &&(newposy<DIM)&&(newposy>=0)){
       if( scacc[newposx][newposy] == 0){
          scacc[newposx][newposy]=mossa;
          ret=muovi(mossa+1,newposx,newposy,scacc);
          if(ret == 0)
             scacc[newposx][newposy]=0;
          else
             return 1;
          }
    }
 }
return 0;
}
```

Ricorsione e tecniche algoritmiche

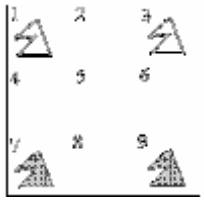
La risoluzione di un problema necessita di una fase preliminare di analisi volta alla ricerca di una qualche analogia, di una caratteristica strutturale che ci permetta di formularlo (e risolverlo) in maniera efficiente.

Si consideri ad esempio il problema di dover scambiare di posto i due cavalli bianchi in alto con i due neri in basso, nella seguente scacchiera usando la caratteristica mossa ad *elle* dei due pezzi:



Il problema può essere risolto in maniera bruta (brute force method) andando a considerare tutti i possibili casi ma….

La ricorsione non è sempre utile…..



Una soluzione semplice ed efficiente può essere ottenuta osservando gli spostamenti che nel nostro particolare caso come mostrato in figura, presentano una qualche regolarità.

Ad esempio il cavallo bianco in posizione 1 può spostarsi in posizione 6; e se spostiamo il cavallo nero in posizione 7 alla posizione 2, il cavallo bianco può ulteriormente spostarsi in posizione 7…. per poi proseguire…..

A questo punto la soluzione del problema è banale: basta far circolare i pezzi in senso orario (o antiorario).

Passeggiando a Manhattan..

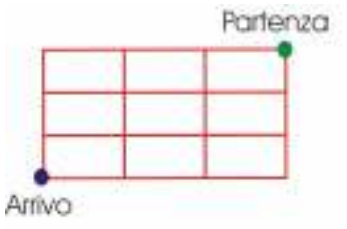
Si immagini di voler fare una passeggiata tra i blocks di Manhattan. La rete stradale di questo quartiere di New York è molto particolare, perché in vaste aree dà luogo a una griglia molto regolare:

Si supponga, ad esempio, di partire a piedi dall'incrocio della 73rd St. con la Lexington Av., per raggiungere l'incrocio della 70th St. con la 5th Ev.:

Manhattan

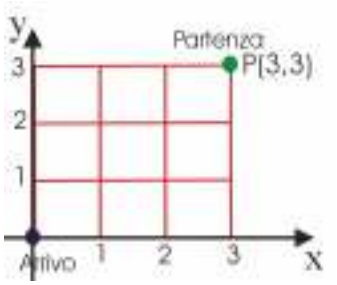
Quanti sono i possibili cammini di lunghezza minima tra l'incrocio di partenza e quello di arrivo?

Innanzitutto è sufficiente considerare la porzione di rete stradale che delimita il più piccolo rettangolo contenente l'incrocio di partenza e quello di arrivo.



Manhattan

La griglia può essere immersa in un piano cartesiano, scegliendo le unità degli assi di modo che la meta sia l'origine ***O(0,0)*** del riferimento e gli altri incroci della rete abbiano coordinate intere e consecutive.

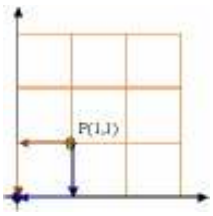


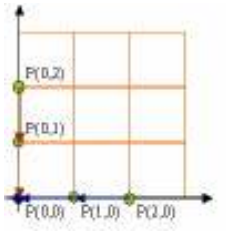
Il problema "Manhattan" può quindi essere espresso nel seguente modo: dato un punto ***P(m,n)*** di coordinate intere ***m*** ed ***n***, determinare il *numero* di spezzate di lunghezza minima ***M(m,n)*** che congiungono ***P*** ad ***O(0,0)***, lungo il reticolo dei punti di coordinate intere.

Manhattan

Osserviamo innanzitutto cosa succede lungo i punti del reticolo vicino all'origine.

M(1,3)=M(3,1)=4
M(1,2)=M(2,1)=3
M(1,1)=2;





Possiamo assumere che per i punti lungo l'asse ***M(0,n)=M(m,0)=1***

Manhattan

Ogni cammino può essere modellato come sequenza di istruzioni elementari di spostamento nelle direzioni cardinali. Possiamo caratterizzare i cammini cercati attraverso le seguenti congetture:

- I cammini minimi sono fatti da spostamenti elementari solo nelle direzioni Sud ed Ovest;
- Se *P(m,n)* è il punto di partenza, allora la lunghezza dei cammini è esattamente ***d(P,O)=m+n***.

Coincidenze

Facendo corrispondere ad ogni incrocio di partenza della nostra rete di 4x4 incroci il corrispondente valore M(m,n), si trova la tabella di numeri

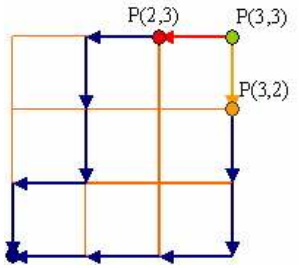
1	4	10	20
1	3	6	10
1	2	3	4
1	1	1	1

che letta in diagonale nella direzione Nord-Est, si scopre essere parte del Triangolo di Pascal…….

Soluzione…

Il numero di cammini minimi da ***P(m,n)*** sono la somma dei cammini minimi dei due incroci adiacenti in direzione Sud ed Ovest, in formule:

$$M(m,n)=M(m,n-1)+M(m-1,n), \text{ con } m,n>0$$



Esercizi

3. Dato un carattere, eseguire (ricorsivamente):
   a) il controllo della sua appartenenza ad una stringa;
   b) il conteggio di tutte le sue occorrenze in una stringa;
   c) le rimozione di tutte le sue occorrenze in una stringa.

4. Scrivere per una sottostringa i metodi equivalenti a quelli dell' esercizio precedente.

Esercizio 3

```
public static boolean isInString(char c, String s)
  { //quesito a)
    for (int i=0; i != s.length(); i++)
       if (c == s.charAt(i))
          return true;
    return false;
  }
```

```
public static int countInString(char c, String s)
  {int count = 0;        //quesito b)
   for (int i=0; i != s.length(); i++)
       if (c == s.charAt(i)) count++;
    return count;
  }
```

Esercizio 3

```
public  static  String  getStringWithoutChar(char  c,
  String s)
  {                      //quesito c)
    String ns="";
    for (int i=0; i != s.length(); i++)
       if (! (c == s.charAt(i)))
          ns= ns + s.charAt(i);
    return ns;
  }
```

Esercizio 3

```
public  static  String  getStringWithoutChar1(char  c,
  String s)
 {//soluzione alternativa quesito c)
  for (int i=0; i != s.length(); i++)
   if (c == s.charAt(i))
      s=s.substring(0,i)+s.substring(i+1,s.length());
  return s;
    }
```

Esercizio 4

```
public static int isInString(String c, String s)
  { //quesito a
    if (c.length()!= 0 && s.length()!= 0)

      if (c.charAt(0) == s.charAt(0))
         return isInString(c.substring(1),s.substring(1));
      else return isInString(c,s.substring(1));

    else if (c.length()==0) return s.length();
         else return -1;
              }
```

Esercizio 4

```
public static int countInString(String c, String s)
  { // quesito b
    int count = 0;
    int i=0;
    while( i != s.length()){
      int r = isInString(c, s.substring(i));
      if (r!=-1) {//r è la lungh. del resto di s[i..]
                  //dopo la prima occorrenza di c
                  //-1 se c non è' sottostringa
        count++;
        i = s.length()-r;
                        //Ci posizioniamo subito dopo
                        //la prima occorrenza
      }
      else return count;
    }
    return count;
  }
```

Esercizio 4

```
public static String getStringWithout(String c, String s)
  { //quesito c
    String ns="";
    int i=0;
    while( i != s.length()){
      int r = isInString(c, s.substring(i));
      if (r!=-1) { //r è la lungh. del resto di s[i..]
                  // dopo la prima occorrenza di c
                  //-1 se c non è' sottostringa
         ns= ns+s.substring(i,s.length() - r-c.length());
         i = s.length() - r;
                  //Ci posizioniamo subito dopo
                  // la prima occorrenza
       }
      else return ns+s.substring(i);
      }
    return ns;
  }
```