Lezione XXI
Lu 12-Nov-2007

Ricorsione

Il calcolo del fattoriale

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

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

dove **n** è un numero intero non negativo

Il calcolo del fattoriale

- Vediamo di capire cosa significa…

$0! = 1$

$1! = 1(1-1)! = 1 \cdot 0! = 1 \cdot 1 = 1$

$2! = 2(2-1)! = 2 \cdot 1! = 2 \cdot 1 = 2$

$3! = 3(3-1)! = 3 \cdot 2! = 3 \cdot 2 \cdot 1 = 6$

$4! = 4(4-1)! = 4 \cdot 3! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$

...

- 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

- Fin qui, nulla di nuovo… però abbiamo dovuto fare un'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);
}
```

- Si potrebbe pensare: "Non è possibile ***invocare un metodo*** mentre si esegue ***il metodo stesso***!"
- Invece, come è facile verificare scrivendo un programma che usi **factorial()**, questo è lecito in Java, così come è lecito in quasi tutti i linguaggi di programmazione

La ricorsione

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

  ricorsione

  e un metodo che ne faccia uso si chiama

  metodo ricorsivo

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

La ricorsione

- Per capire come utilizzare correttamente ***la ricorsione***, vediamo innanzitutto ***come funziona***
- Quando un metodo ricorsivo invoca se stesso, ***la macchina virtuale Java esegue le stesse azioni che vengono eseguite quando viene invocato un metodo qualsiasi***
  - *sospende* l'esecuzione del metodo invocante
  - *esegue il metodo invocato* fino alla sua terminazione
  - *riprende* l'esecuzione del metodo invocante dal punto in cui era stata sospesa

La ricorsione

- Vediamo la sequenza usata per calcolare 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 che poi si accorcia fino ad estinguersi

La ricorsione: caso base

- **Prima regola**
  - il metodo ricorsivo ***deve fornire la soluzione del problema in almeno un caso particolare***, ***senza ricorrere a una chiamata ricorsiva***
  - tale caso si chiama ***caso base*** della ricorsione
  - nel nostro esempio, il caso base è

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

  - a volte ci sono più casi base, non è necessario che sia unico

La ricorsione: passo ricorsivo

- **Seconda regola**
  - il metodo ricorsivo ***deve effettuare la chiamata ricorsiva dopo aver semplificato il problema***
  - nel nostro esempio, per il calcolo del fattoriale di **n** si invoca la funzione ricorsivamente per conoscere il fattoriale di **n-1**, cioè per risolvere un problema più semplice

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

  - il concetto di "problema più semplice" varia di volta in volta: in generale, ***bisogna tendere a un caso base***

La ricorsione: un algoritmo?

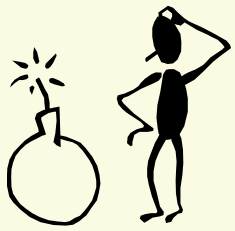
- Le regole appena viste sono fondamentali per poter dimostrare che la soluzione ricorsiva di un problema sia un algoritmo
  - in particolare, che arrivi a conclusione in un numero **finito** di passi
- Si potrebbe pensare che le chiamate ricorsive si possano succedere una dopo l'altra, all'infinito

La ricorsione: un algoritmo?

- Invece, se
  - a ogni invocazione il problema diventa sempre più semplice e si avvicina al caso base
  - la soluzione del caso base non richiede ricorsione

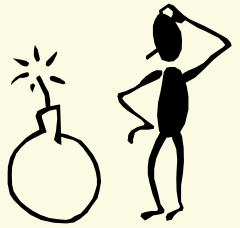
allora certamente la soluzione viene calcolata in un numero finito di passi, per quanto complesso possa essere il problema

Ricorsione infinita



- Abbiamo visto che non tutti i metodi ricorsivi realizzano algoritmi
  - ***se manca il caso base***, il metodo ricorsivo continua ad invocare se stesso all'infinito
  - ***se il problema non viene semplificato a ogni invocazione ricorsiva***, il metodo ricorsivo continua ad invocare se stesso all'infinito
- Dato che la lista dei metodi "in attesa" si allunga indefinitamente, l'ambiente runtime esaurisce la memoria disponibile per tenere traccia di questa lista e il programma termina con un errore

Ricorsione infinita



- Provare a eseguire un programma che presenta ricorsione infinita

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

- Il programma terminerà con la segnalazione dell'eccezione **StackOverflowError**
  - il ***runtime stack*** (o Java Stack) è la struttura dati all'interno dell'interprete Java che gestisce le invocazioni in attesa
  - ***overflow*** significa ***trabocco***

La ricorsione in coda

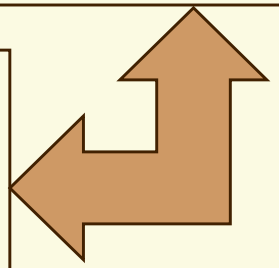
- Esistono diversi tipi di ricorsione
- Il modo visto fino a ora si chiama **ricorsione in coda** (***tail recursion***)
  - il metodo ricorsivo esegue ***una sola invocazione ricorsiva*** e tale invocazione è l'***ultima azione*** del metodo

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

Eliminazione della ricorsione in coda

- La ricorsione in coda può sempre essere agevolmente ***eliminata***, trasformando il metodo ricorsivo in un metodo che usa 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 k = n;
    int p = 1;
    while (k > 0)
    {   p = p * k;
        k--;
    }
    return p;
}
```

La ricorsione in coda

- Allora, a cosa serve la ricorsione in coda?
- Non è necessaria, però in alcuni casi rende il codice più leggibile
- È 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

La ricorsione non in coda

- Se la ricorsione non è in coda, non è facile eliminarla (cioè scrivere codice non ricorsivo equivalente), però si può dimostrare che ciò è sempre possibile
  - deve essere così, perché il processore esegue istruzioni in sequenza e non può tenere istruzioni in attesa… quindi l'interprete deve farsi carico di eliminare la ricorsione (usando il runtime stack)

La ricorsione multipla

- Si parla di ricorsione multipla quando un metodo invoca se stesso più volte durante la sua esecuzione
  - la ricorsione multipla è ancora più difficile da eliminare, ma è sempre possibile
- Esempio: il calcolo dei numeri di Fibonacci

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

I numeri di Fibonacci

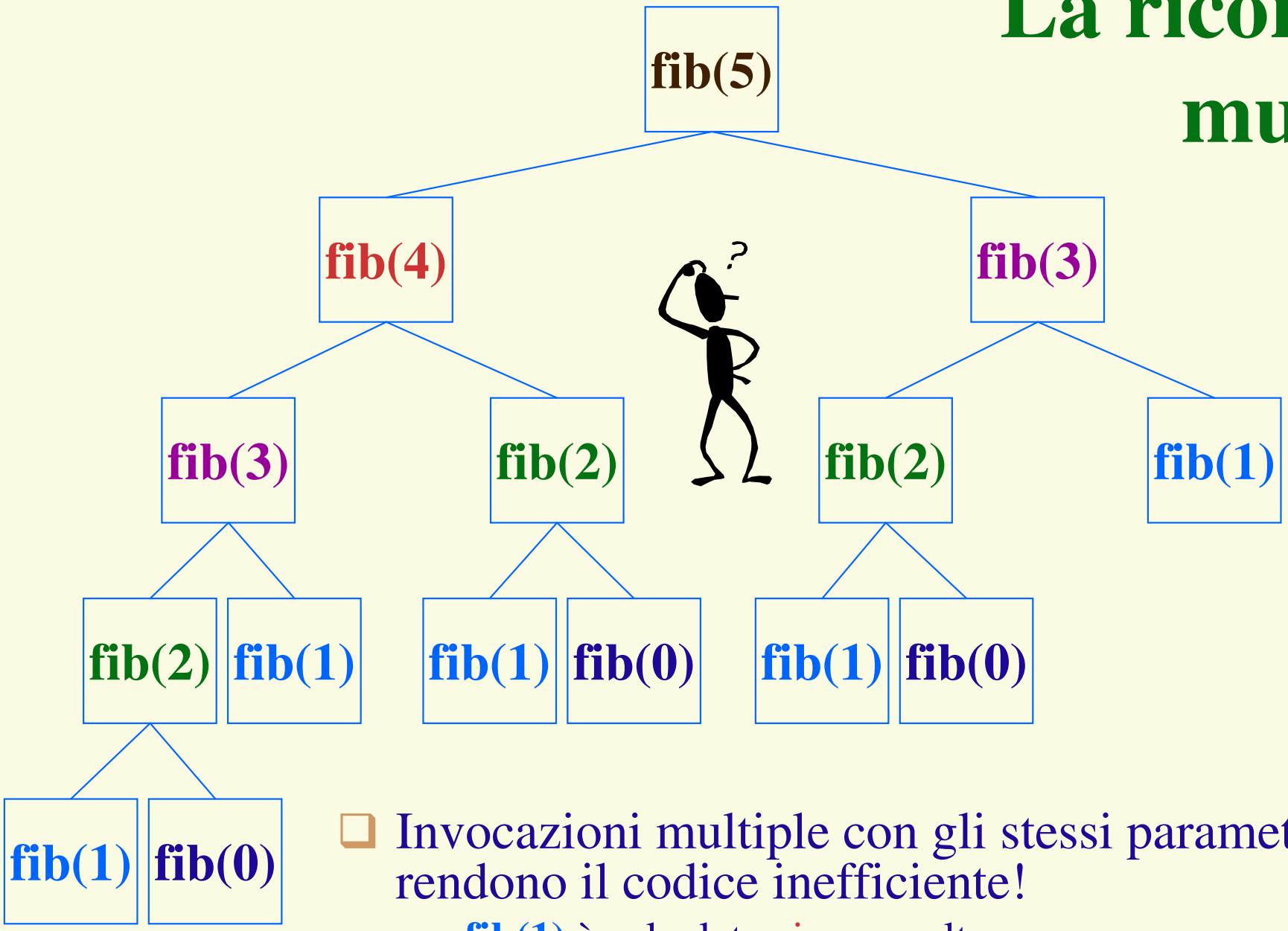
```
public static int fib(int n)
{   if (n < 0)
       throw new IllegalArgumentException();

    if (n < 2)
       return n;

    return fib(n-2) + fib(n-1);
}
```

- Il metodo `fib()` realizza una ricorsione multipla

La ricorsione multipla

- La ricorsione multipla va usata con molta attenzione, perché può portare a programmi molto inefficienti
- Eseguendo il calcolo dei numeri di Fibonacci di ordine crescente...
  - … si nota che il tempo di elaborazione cresce MOLTO rapidamente… quasi 3 milioni di invocazioni per calcolare fib(31) !!!!
  - più avanti quantificheremo questo problema

La ricorsione multipla



- Invocazioni multiple con gli stessi parametri rendono il codice inefficiente!
  - **fib(1)** è calcolata cinque volte
  - **fib(2)** è calcolata tre volte
  - **fib(3)** è calcolata due volte

Esempio di algoritmo ricorsivo Inversione di una stringa

- Si inverta l'ordine dei caratteri di una stringa con procedimento ricorsivo
  - `"uno" -> "onu"`
- Algoritmo ricorsivo:
  - **precondizioni**: se la stringa s da invertire non e' una stringa, ma il riferimento null, si restituisca null
  - **caso base**: se il numero di caratteri della stringa s e' minore di 2, si restituisca la stringa perche' non serve invertire
  - Se non siamo nei casi base, si estragga il primo carattere della stringa s[0]. Si inverta ricorsivamente la sottostringa con indici da 1 a s.length – 1. Si restituisca la concatenazione della sottostringa invertita col primo carattere.
  - Esempio: s = "uno"; si ritorna "on" + "u"

think recursively!

Esempio di algoritmo ricorsivo

```
private static String reverseString(String s)
{
   // precondizioni
   if (s == null)
      return null;

   // caso base
   if (s.length() < 2)
      return s;

   // passo ricorsivo
   return reverseString(s.substring(1))
                                + s.charAt(0);
}
```

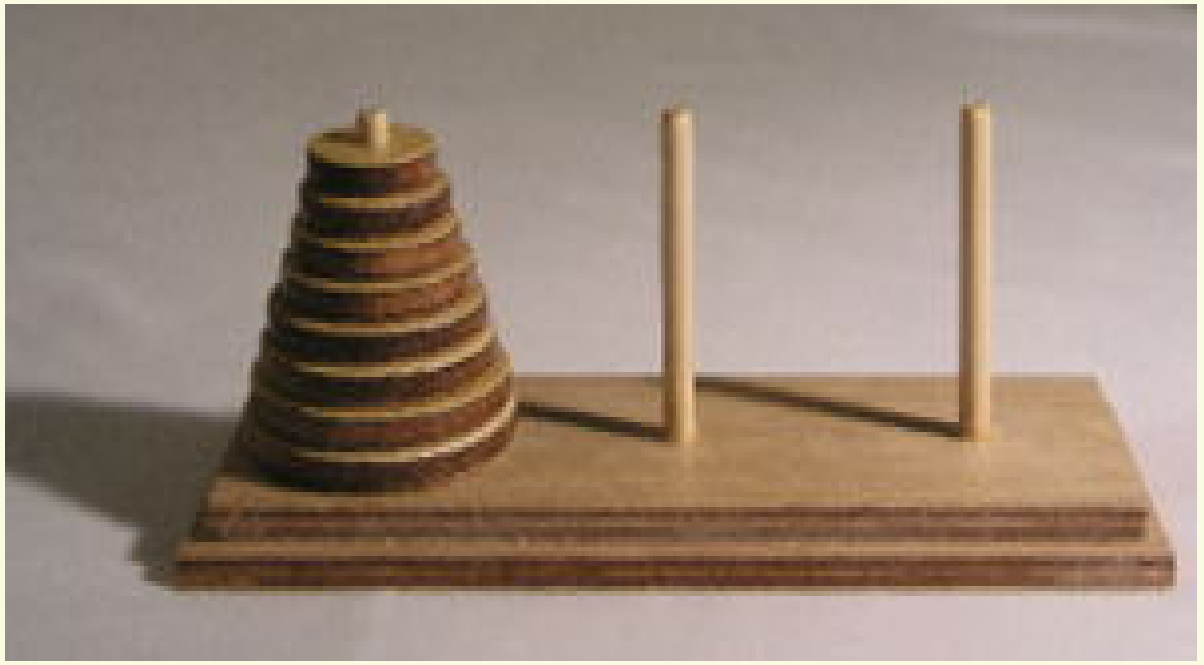
Stringa che contiene tutti i caratteri di s eccetto il primo

Ricorsione: Torri di Hanoi

http://www.mazeworks.com/hanoi/index.htm

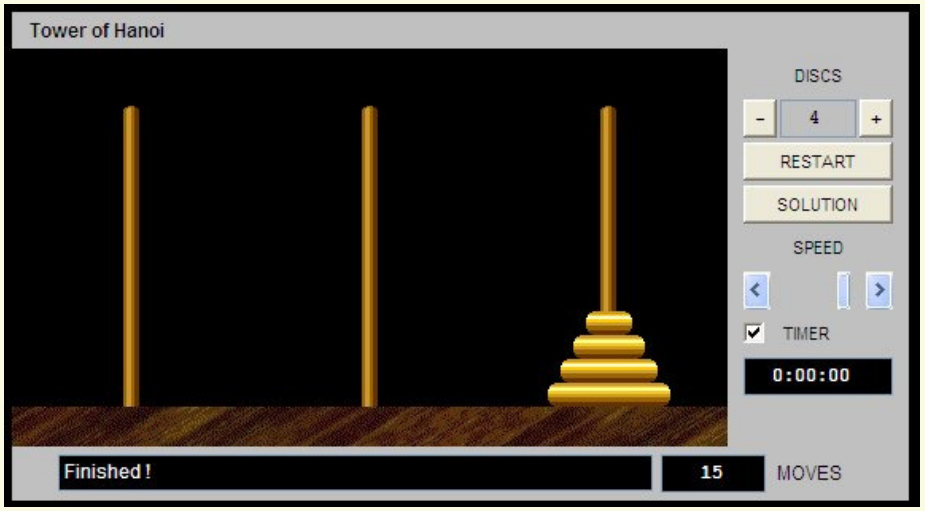
Torri di Hanoi: regole

- Il rompicapo è costituito da tre pile di dischi ("torri") allineate
  - all'inizio tutti i dischi si trovano sulla pila di sinistra
  - alla fine tutti i dischi si devono trovare sulla pila di destra
- I dischi sono tutti di dimensioni diverse e quando si trovano su una pila devono rispettare la seguente regola
  - ***nessun disco può avere sopra di sé dischi più grandi***



Torri di Hanoi: regole

Situazione iniziale



Situazione finale

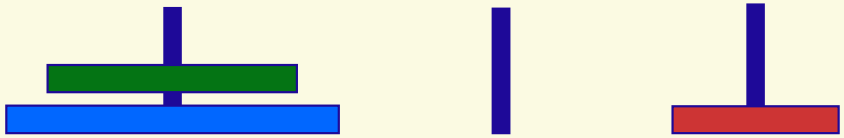
Torri di Hanoi: regole

- Per risolvere il rompicapo bisogna ***spostare un disco alla volta***
  - un disco può essere rimosso dalla cima della torre e inserito in cima a un'altra torre
    - non può essere "parcheggiato" all'esterno…
  - in ogni momento deve essere rispettata la regola vista in precedenza
    - ***nessun disco può avere sopra di sé dischi più grandi***

Torri di Hanoi: n = 3

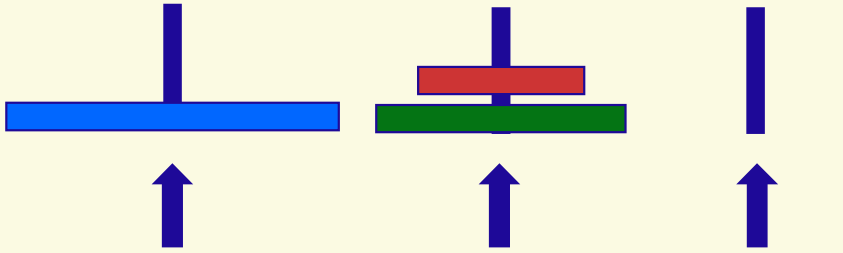
Situazione iniziale

mossa 1



mossa 2

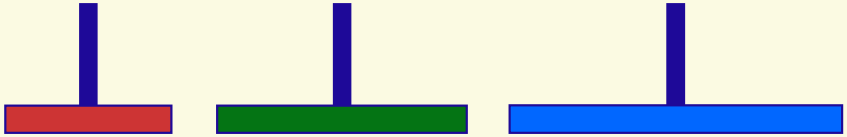
mossa 3



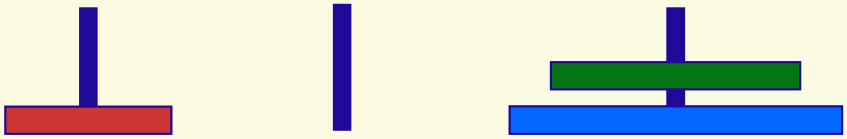
Torre 1 Torre 2 Torre 3

mossa 4

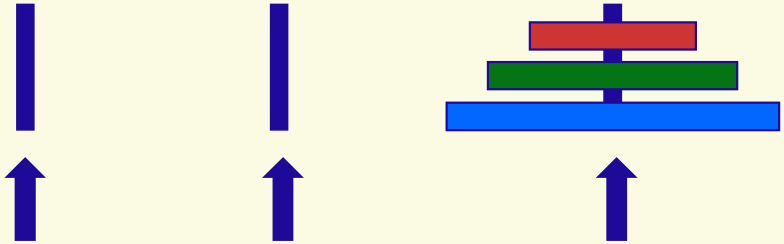
mossa 5



mossa 6



mossa 7 Situazione finale



Torre 1 Torre 2 Torre 3

Algoritmo di soluzione

- Per il rompicapo delle Torri di Hanoi è noto un algoritmo di soluzione ricorsivo
- Il problema generale consiste nello spostare **n** dischi da una torre a un'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, come sempre si fa nella ricorsione
- Per descrivere l'algoritmo identifichiamo le torri con i numeri interi 1, 2 e 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 nella torre 1 viene portato 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** vale 1 e l'algoritmo usa il solo passo **2**, che non ha chiamate ricorsive

Algoritmo ricorsivo

- Si può dimostrare che il numero di mosse necessarie per risolvere il rompicapo con l'algoritmo proposto è

$$2^n - 1$$

- con n pari al numero di dischi

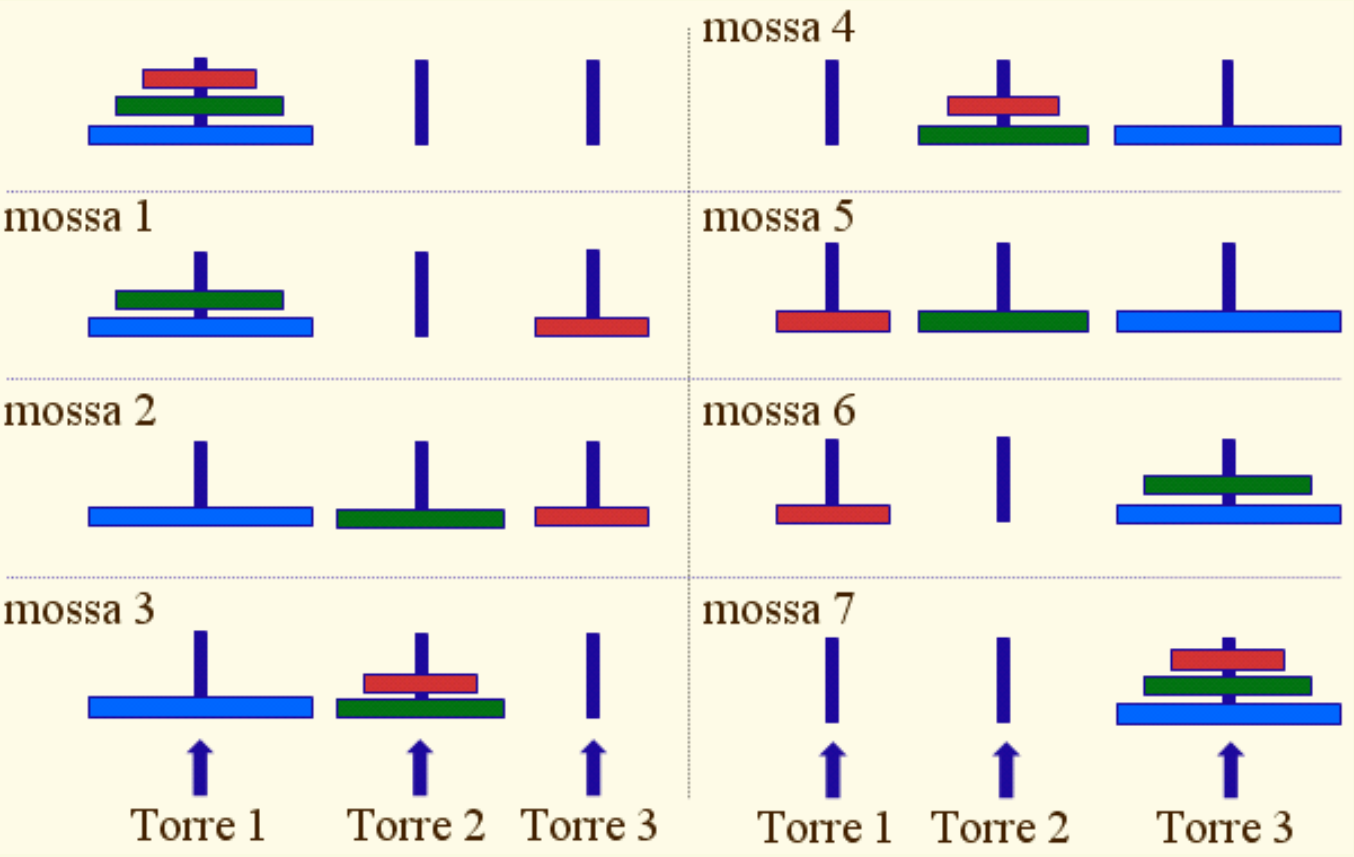
Soluzione delle Torri di Hanoi

```
public class HanoiSolver
{  public static void main(String[] v)
   {  System.out.println("Soluzione della torre"
         + " di Hanoi per " + v[0] + " dischi");
      solveHanoi(1,3,2,Integer.parseInt(v[0]));
   }
   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 T" + from + " a T" + to);
         solveHanoi(temp, to, from, dim-1);
      }
   }
}
```

Torri di Hanoi: n = 2

```
main() chiama solveHanoi(1,3,2,2)

Soluzione per la Torre di Hanoi con 2 dischi
start solveHanoi(1, 3, 2, 2)
   start solveHanoi(1, 2, 3, 1)
      start solveHanoi(1, 3, 2, 0)
      stop  solveHanoi(1, 3, 2, 0)
   Sposta da T1 a T2
      start solveHanoi(3, 2, 1, 0)
      stop  solveHanoi(3, 2, 1, 0)
   stop  solveHanoi(1, 2, 3, 1)
Sposta da T1 a T3
   start solveHanoi(2, 3, 1, 1)
      start solveHanoi(2, 1, 3, 0)
      stop  solveHanoi(2, 1, 3, 0)
   Sposta da T2 a T3
      start solveHanoi(1, 3, 2, 0)
      stop  solveHanoi(1, 3, 2, 0)
   stop  solveHanoi(2, 3, 1, 1)
stop  solveHanoi(1, 3, 2, 2)
```

Torri di Hanoi: n = 3



Soluzione della Torre di Hanoi con 3 dischi

Sposta da T1 a T3

Sposta da T1 a T2

Sposta da T3 a T2

Sposta da T1 a T3

Sposta da T2 a T1

Sposta da T2 a T3

Sposta da T1 a T3

Quando finirà il mondo?

Torri di Hanoi: la leggenda

- Una leggenda narra che alcuni monaci buddisti in un tempio dell'Estremo Oriente siano da sempre impegnati nella soluzione del rompicapo, spostando fisicamente i loro **64** dischi da una torre all'altra, consapevoli che quando avranno terminato il mondo finirà
- Sono necessarie $2^{64}-1$ mosse, che sono circa 16 miliardi di miliardi di mosse… cioè circa 1.6×10^{18}
- Supponendo che i monaci facciamo una mossa ogni minuto, essi fanno circa 500 000 mosse all'anno: quindi il mondo finirà tra circa 30 mila miliardi di anni…
- Un processore a 1GHz che fa una mossa a ogni intervallo di clock (un miliardo di mosse al secondo…) impiega 16 miliardi di secondi, che sono circa 500 anni...

Esempio di programma ricorsivo

Esercizio: Permutazioni

- Vogliamo risolvere il problema di determinare tutte le possibili permutazioni dei caratteri presenti in una stringa
  - facciamo l'ipotesi che non ci siano caratteri ripetuti
- Ricordiamo dalla matematica combinatoria che il numero di permutazioni di *N* simboli è *N!*
- Esempio: **ABC**
  - **ABC ACB BAC BCA CAB CBA**

Esercizio: Permutazioni

- Esiste un semplice algoritmo ricorsivo per trovare le permutazioni di una stringa di lunghezza N
- Se N vale 1, l'unica permutazione è la stringa stessa
- Altrimenti
  - estrai il primo carattere dalla stringa
  - calcola le (N-1)! permutazioni della stringa rimanente
  - per ognuna delle permutazioni (incomplete) ottenute
    - genera N permutazioni (complete) inserendo il carattere precedentemente estratto in ognuna delle posizioni possibili nella permutazione incompleta

Esercizio: Permutazioni

- Esempio: uso l'algoritmo per generare le permutazioni della stringa "**ABC**"
  - Estraggo il primo carattere: '**A**'
  - Calcolo tutte le permutazioni di (n-1) caratteri della stringa rimanente: "BC"
    - "**BC**" "CB"
  - per ognuna delle permutazioni (incomplete) ottenute genera N permutazioni (complete) inserendo il carattere precedentemente estratto in ognuna delle posizioni possibili nella permutazione incompleta
    - "**ABC**" "**BAC**" "**BCA**" "**ACB**" "**CAB**" "**CBA**"

Esercizio: Permutazioni

- Analizziamo meglio questo punto
  - calcola le (N-1)! permutazioni della stringa rimanente
  - per ognuna delle permutazioni (incomplete) ottenute
    - genera N permutazioni (complete) inserendo il carattere precedentemente estratto in ognuna delle posizioni possibili nella permutazione incompleta
- Le posizioni in cui si può inserire un carattere in una delle permutazioni incomplete, che ha dimensione N-1, sono le N-2 posizioni che si trovano tra due caratteri, più la posizione iniziale e la posizione finale
  - sono quindi N posizioni diverse

```
public static String[] permutations(String p)
{  // gestiamo i casi base
   if (p == null || p.length() == 0)
      return new String[0]; // oppure return null
   if (p.length() == 1) return new String[] {p};
   // isoliamo il primo carattere
   String c = p.substring(0,1);
   // passo ricorsivo: permutazioni incomplete
   String[] cc = permutations(p.substring(1));
   // numero di permutazioni da generare
   String[] r = new String[p.length() * cc.length];
   for (int i = 0; i < p.length(); i++)
      for (int j = 0; j < cc.length; j++)
      {  String s = cc[j];
         String sLeft = s.substring(0,i);
         String sRight = s.substring(i);
         r[i*cc.length+j] = sLeft + c + sRight;
      }
   return r;
}
```

Esempio: Permutazioni

- Alcuni commenti generali sulla gestione delle condizioni "eccezionali"
  - il metodo riceve un riferimento a una stringa: tale riferimento può essere **null** oppure la stringa può essere vuota (cioè avere lunghezza zero)
    - in entrambi i casi il metodo ricorsivo non funzionerebbe correttamente, quindi li inseriamo come casi base della ricorsione
  - cosa restituiamo?
    - quando un metodo deve restituire un oggetto, nei casi in cui riceve un parametro non corretto di solito restituisce **null**

```
if (p == null || p.length() == 0)
   return null;
```

Esempio: Permutazioni

```
if (p == null || p.length() == 0)
    return new String[0];
```

- In questo caso il metodo deve restituire un oggetto di tipo un po' particolare, in quanto è un array
- Sarebbe comunque corretto restituire **null**, ma di solito si preferisce restituire un array di lunghezza zero (perfettamente lecito), in modo che il metodo invocante riceva un array valido, seppure vuoto
- In questo modo, il codice seguente funziona…

```
String[] x = permutations("");
for (int i = 0; i < x.length; i++)
    System.out.println(x[i]);
```

Esempio: Permutazioni

```
if (p == null || p.length() == 0)
    return new String[0];
```

- Notiamo che anche ***l'ordine in cui vengono valutate le due condizioni è molto importante***
- Se **p** è **null**, la prima condizione è vera e per la strategia di cortocircuito nella valutazione dell'operatore `||` ***la seconda condizione non viene valutata***
  - se venisse valutata, verrebbe lanciata l'eccezione **NullPointerException** !!

Esempio: Permutazioni

```
// isoliamo il primo carattere
String c = p.substring(0,1);
// passo ricorsivo
String[] cc = permutations(p.substring(1));
// numero di permutazioni da generare
String[] r = new String[p.length() *
    cc.length];
```

- Per calcolare la dimensione del vettore che conterrà le permutazioni, sfruttiamo le informazioni ottenute dall'invocazione ricorsiva
  - il numero di permutazioni è uguale alla dimensione della stringa moltiplicata per il numero di permutazioni (incomplete) già generate
    - n = p.length(), (n-1) ! =cc.length
    - p.length * cc.length = n * (n-1) !

Esempio: Permutazioni

```
for (int i = 0; i < p.length(); i++)
   for (int j = 0; j < cc.length; j++)
   {  String s = cc[j];
      String sLeft = s.substring(0,i);
      String sRight = s.substring(i);
      r[i*cc.length+j] = sLeft + c + sRight;
   }
```

- Per completare le permutazioni inseriamo il carattere **c** in tutte le posizioni possibili in ciascuna permutazione incompleta **s**
- Per ogni **s**, calcoliamo le sottostringhe che verranno concatenate a sinistra e a destra di **c**
- Sfruttiamo il fatto che il metodo **substring()** gestisce in modo congruente le situazioni "anomale"

Esempio: Permutazioni

```
String sLeft = s.substring(0,i);
```

- Quando **i** vale **0**, **substring** restituisce una stringa vuota, che è proprio ciò che vogliamo

```
String sRight = s.substring(i);
```

- Quando **i** vale **p.length( )-1** (suo valore massimo), allora **i** è anche uguale a **s.length( )**
  - in questo caso particolare, **substring()** non lancia eccezione, ma restituisce una stringa vuota, che è proprio ciò che vogliamo

Esempio: Permutazioni

```
r[i*cc.length+j] = sLeft + c + sRight;
```

- Analizziamo meglio questa espressione dell'indice
- Globalmente, tale indice deve andare da `0` a `p.length() * cc.length` (escluso)
- Verifichiamo innanzitutto i limiti
  - per `i = 0` e `j = 0`, l'indice vale `0`
  - per `i = p.length()-1` e `j = cc.length-1`, l'indice vale

```
(p.length()-1)*cc.length + cc.length - 1
=p.length()*cc.length - 1
```

(come volevamo)

Esercizio: Permutazioni

```
r[i*cc.length+j] = sLeft + c + sRight;
```

- Alla prima iterazione di `i`, l'indice varia tra `0` e `cc.length-1` (perché `i` vale 0)
- Alla seconda iterazione di `i`, l'indice varia tra `1*cc.length+0 = cc.length` e `1*cc.length+cc.length-1 = 2*cc.length-1`
- Si osserva quindi che gli indici vengono generati consecutivamente, senza nessun valore mancante e senza nessun valore ripetuto

Esercizio: Permutazioni

```
public class PermutationGenerator
{
   ... // metodo permutations

   public static void main(String[] args)
   {
      if (args.length < 1)
      {  System.out.println(
           "uso: $ java PermutationGenerator string");
         return;
      }

      System.out.println(
        "Permutazioni della stringa " + args[0]);

      String[] permutations = permutations(args[0]);

      for (int i = 0; i < permutations.length; i++)
         System.out.println(permutations[i]);
   }
}
```

Lezione XXII
Ma 13-Nov-2007

Ordinamento

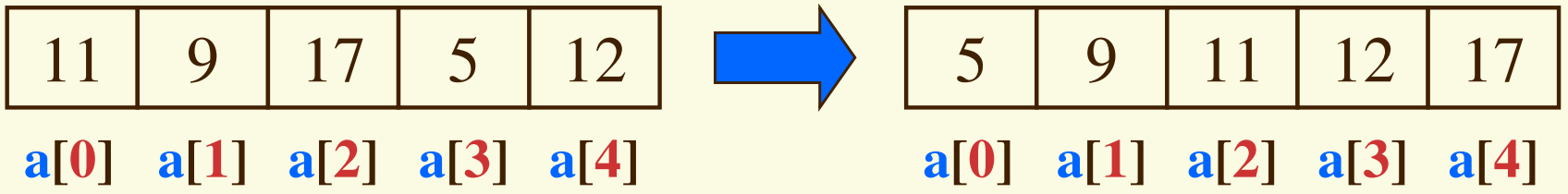
Motivazioni

- Un problema davvero molto frequente nella elaborazione dei dati consiste nell'***ordinamento*** dei dati stessi, secondo un criterio prestabilito
  - nomi in ordine alfabetico, numeri in ordine crescente…
- Studieremo diversi algoritmi per l'ordinamento, introducendo anche ***uno strumento analitico per la valutazione delle loro prestazioni***
- Su dati ordinati è poi possibile ***fare ricerche in modo molto efficiente***, e studieremo algoritmi adeguati
- Gli algoritmi che studieremo sono ricorsivi ed efficienti, ma ***non tutti gli algoritmi ricorsivi sono efficienti***...

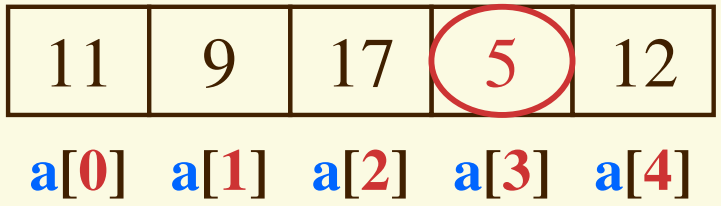
Ordinamento per selezione

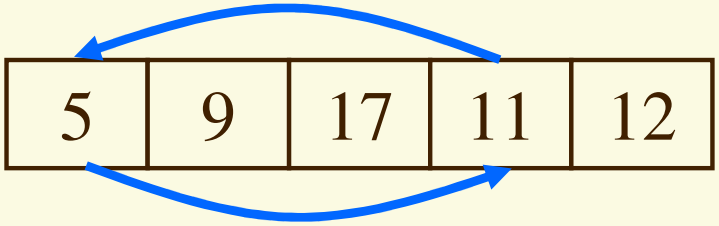
Ordinamento per selezione

- Per semplicità, analizzeremo prima gli algoritmi per ***ordinare un insieme di numeri*** (interi) memorizzati in un array
  - vedremo poi come si estendono semplicemente al caso in cui si debbano elaborare oggetti anziché numeri
- Prendiamo in esame un array **a** da ordinare in senso crescente



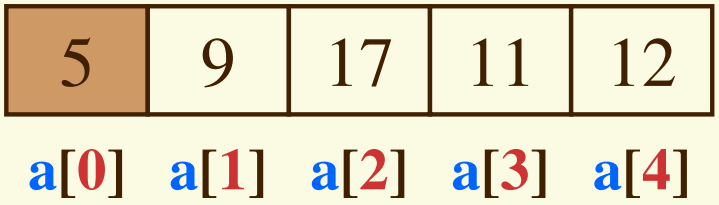
Ordinamento per selezione





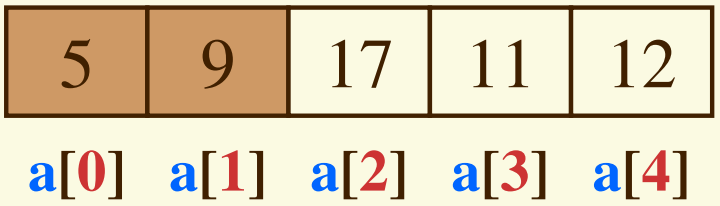
- Per prima cosa, bisogna trovare ***l'elemento minore presente nell'array***, come sappiamo già fare
  - in questo caso è il numero **5** in posizione **a[3]**
- Essendo l'elemento minore, la sua posizione corretta nell'array ordinato è **a[0]**
  - in **a[0]** è memorizzato il numero **11**, da spostare
  - non sappiamo quale sia la posizione corretta di **11**
    - lo spostiamo temporaneamente in **a[3]**
  - quindi, ***scambiamo*** **a[3]** con **a[0]**

Ordinamento per selezione

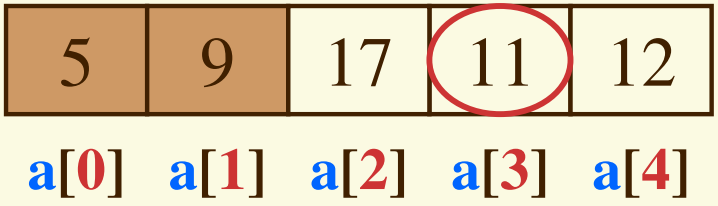


la parte colorata dell'array è già ordinata

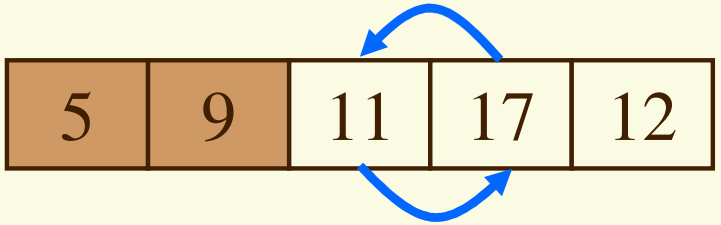
- La parte sinistra dell'array è già ordinata e non sarà più considerata, dobbiamo ordinare la parte destra
- Ordiniamo la parte destra ***con lo stesso algoritmo***
  - cerchiamo l'elemento minore, che è **9** in posizione **a[1]**
  - dato che ***è già nella prima posizione*** **a[1]** ***della parte da ordinare***, ***non c'è bisogno di fare scambi***

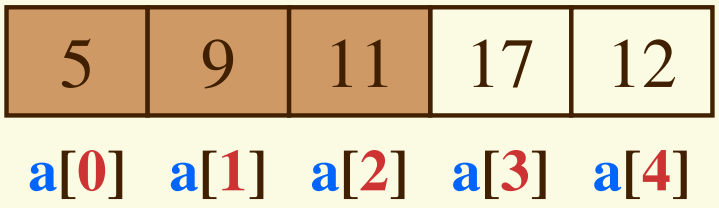


Ordinamento per selezione

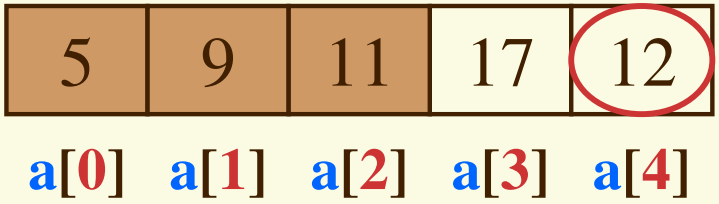


- Proseguiamo per ordinare la parte di array che contiene gli elementi **a[2]**, **a[3]** e **a[4]**
  - l'elemento minore è il numero **11** in posizione **a[3]**
  - scambiamo **a[3]** con **a[2]**

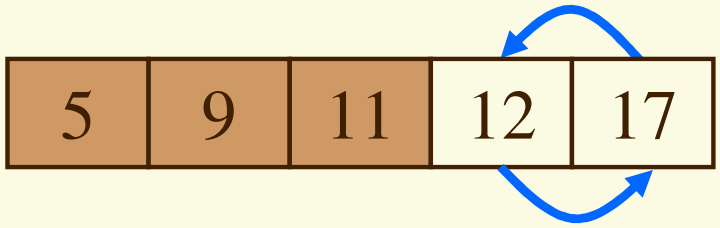


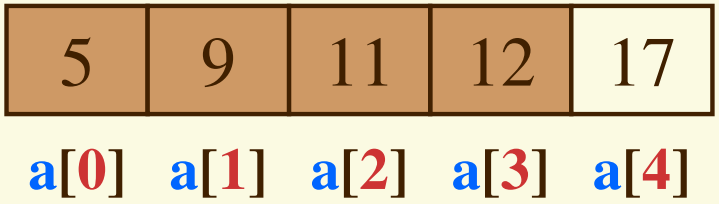


Ordinamento per selezione



- Ora l'array da ordinare contiene **a[3]** e **a[4]**
  - l'elemento minore è il numero **12** in posizione **a[4]**
  - scambiamo **a[4]** con **a[3]**



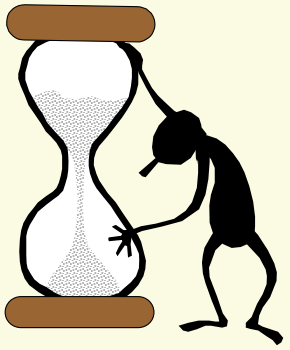


- A questo punto ***la parte da ordinare contiene un solo elemento***, quindi ***è ovviamente ordinata***

```
public class ArrayAlgorithms
{  public static void selectionSort(int[] a)
   {  for (int i = 0; i < a.length - 1; i++)
      {  int minPos = findMinPos(a, i);
         if (minPos != i) swap(a, minPos, i);
      }
   }
   private static void swap(int[] a, int i, int j)
   {  int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
   private static int findMinPos(int[] a, int from)
   {  int pos = from;  int min = a[from];
      for (int i = from + 1; i < a.length; i++)
      {  int temp = a[i]; // metodo complicato per
         if (temp < min)  // rendere più semplice la
         {  pos = i;      // valutazione delle
            min = temp;   // prestazioni (più avanti)
         }
      }
      return pos;
   }
}
```

Ordinamento per selezione

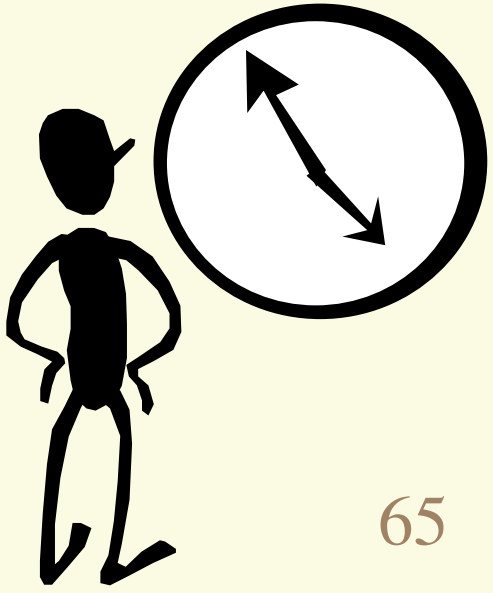
- L'algoritmo di ordinamento per selezione è corretto ed è in grado di ordinare qualsiasi array
  - allora perché studieremo altri algoritmi di ordinamento?
  - ***a cosa serve avere diversi algoritmi per risolvere lo stesso problema?***
- Vedremo che ***esistono algoritmi di ordinamento che***, a parità di dimensioni del vettore da ordinare, ***vengono eseguiti più velocemente***



Rilevazione delle prestazioni

Rilevazione delle prestazioni

- Per valutare l'efficienza di un algoritmo si misura il tempo in cui viene eseguito su insiemi di dati di dimensioni via via maggiori
- Il tempo ***non va misurato con un cronometro***, perché parte del tempo reale di esecuzione non dipende dall'algoritmo
  - caricamento della JVM
  - caricamento delle classi del programma
  - lettura dei dati dallo standard input
  - visualizzazione dei risultati



Rilevazione delle prestazioni

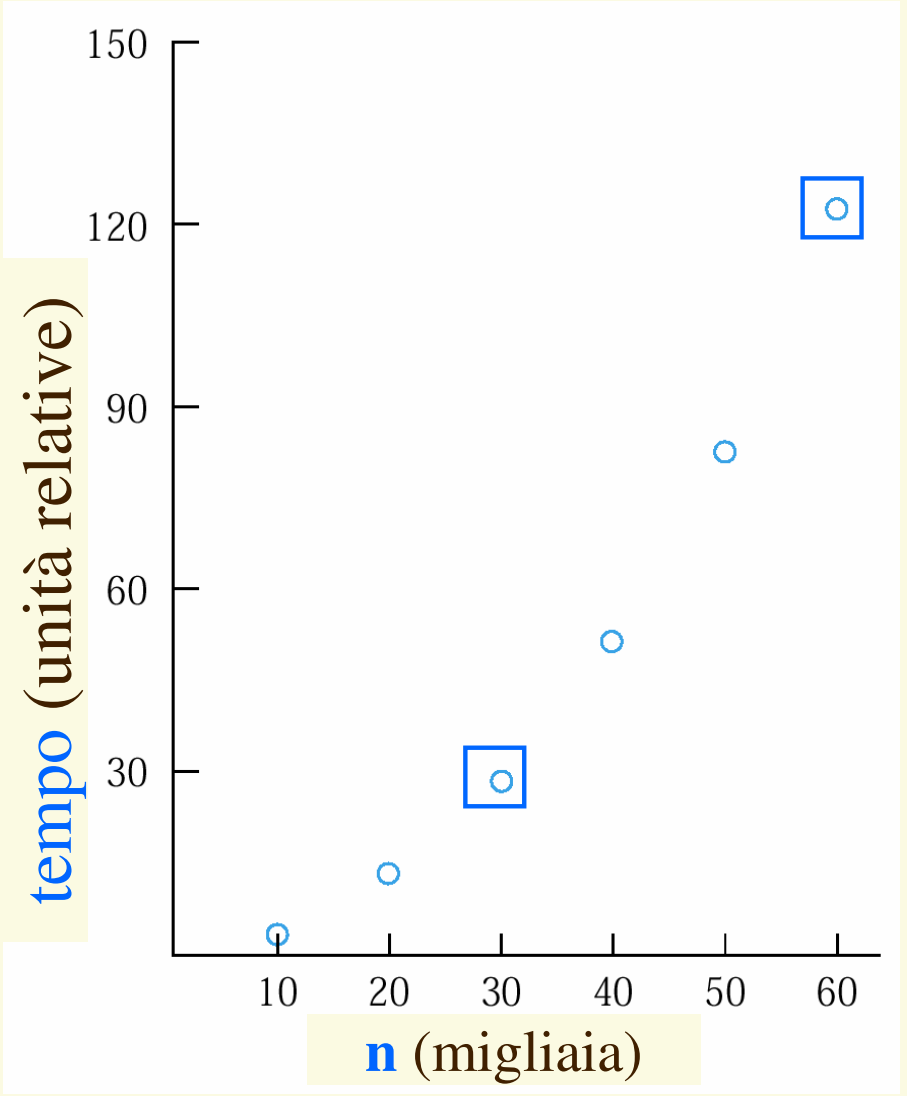
- Il tempo di esecuzione di un algoritmo va misurato all'interno del programma
- Si usa il metodo statico

  `System.currentTimeMillis()`

  che, a ogni invocazione, restituisce un numero di tipo **long** che rappresenta
  - il numero di millisecondi trascorsi da un evento di riferimento (***la mezzanotte del 1 gennaio 1970***)
- Ciò che interessa è la ***differenza*** tra due valori
  - si invoca **System.currentTimeMillis( )** ***prima e dopo*** l'esecuzione dell'algoritmo

Rilevazione delle prestazioni

- Proviamo ad eseguire l'ordinamento di array con diverse dimensioni (**n**), contenenti numeri interi casuali
- Si nota che l'andamento del tempo di esecuzione ***non è lineare***
  - ***se* n *diventa il doppio, il tempo diventa circa il quadruplo!***



Rilevazione delle prestazioni

- Le prestazioni dell'algoritmo di ordinamento per selezione hanno quindi un ***andamento quadratico*** in funzione delle dimensioni dell'array
- Le prossime domande che ci poniamo sono
  - ***è possibile valutare le prestazioni di un algoritmo dal punto di vista teorico?***
    - senza realizzare un programma e senza eseguirlo molte volte, per rilevarne i tempi di esecuzione ed osservarne l'andamento
  - ***esiste un algoritmo più efficiente per l'ordinamento?***

Analisi teorica delle prestazioni

Analisi teorica delle prestazioni

- Il tempo d'esecuzione di un algoritmo dipende dal numero e dal tipo di istruzioni in ***linguaggio macchina*** che vengono eseguite dal processore
- Per fare un'analisi teorica
  - senza realizzare un algoritmo, compilarlo e tradurlo in linguaggio macchina!

  dobbiamo fare delle ***semplificazioni drastiche***
  - ***facciamo un conteggio del numero di accessi in lettura o scrittura a un elemento dell'array***

Analisi teorica delle prestazioni

- Ordiniamo con la selezione un array di `n` elementi
  - per trovare l'elemento minore si fanno `n` accessi
  - per scambiare due elementi si fanno **quattro** accessi
    - trascuriamo il caso in cui non serva lo scambio
  - in totale, `(n+4)` accessi
- A questo punto dobbiamo ordinare la parte rimanente, cioè un array di `(n-1)` elementi
  - serviranno quindi `((n-1) + 4)` accessi

  e via così fino al passo con `n=2`, incluso

Analisi teorica delle prestazioni

- Il conteggio totale degli accessi in lettura è quindi

$$(n+4)+((n-1)+4)+\ldots+(3+4)+(2+4)$$

$$= n+(n-1)+\ldots+3+2 \; + \; (n-1)*4$$

$$= n*(n+1)/2 \; - \; 1 \; + \; (n-1)*4$$

$$= 1/2*n^2 + 9/2*n - 5$$

- Si ottiene quindi ***un polinomio di secondo grado*** in `n`, che giustifica l'andamento ***parabolico*** dei tempi rilevati sperimentalmente

$$n+(n-1)+\ldots+3+2+1 = n*(n+1)/2$$

Andamento asintotico delle prestazioni

$$1/2*n^2 + 9/2*n - 5$$

- Facciamo un'ulteriore semplificazione, tenendo presente che ci interessano le prestazioni per ***valori elevati*** di **n** (***andamento asintotico***)
- Se **n** vale 1000
  - $1/2*n^2$ vale 500000
  - $9/2*n - 5$ vale 4495, circa **1%** del totale

  quindi diciamo che l'andamento asintotico dell'algoritmo in funzione di **n** è $1/2*n^2$

Andamento asintotico delle prestazioni

- Facciamo un'ulteriore semplificazione

  `1/2*n²`

  - ci interessa soltanto valutare cosa succede al tempo d'esecuzione `T(n)` se `n`, ad esempio, raddoppia
  - in questo caso il tempo d'esecuzione *quadruplica*

    `T(n) = 1/2*n²`

    `T(2n) = 1/2*(2n)² = 1/2*4*n²`

    `= 4*T(n)`

- Si osserva che `T(2n) = 4*T(n)`

  anche per `T(n) = n²`

  indipendentemente dal fattore `1/2`

Notazione “O grande”

- Si dice quindi che
  - per ordinare un array con l’algoritmo di selezione si effettua un numero di accessi che è ***dell’ordine*** di n^2
- Per esprimere sinteticamente questo concetto si usa la notazione “O grande” e si dice che il numero degli accessi è $O(n^2)$
- Dopo aver ottenuto una formula che esprime l’andamento temporale dell’algoritmo, si ottiene la notazione “O grande” considerando ***soltanto il termine che si incrementa più rapidamente*** all’aumentare di n, ***ignorando coefficienti costanti***

Notazione “O grande”

- Sia il tempo di calcolo di un algoritmo espresso come T(n), dove N rappresenta la dimensione del problema (ad esempio il numero di elementi di un vettore nel caso di un ordinamento), e sia F(n) una funzione di n
- O grande: diremo che T(n) è **O**(F(n)) se esistono due costanti positive c > 0 e n0 > 0 tali che T(n) <= c F(n) per n > n0
  – *in termini informali T(n) cresce meno di F(n) per n grandi*
- Omega grande: diremo che T(n) è Ω(F(n) se esistono due costanti positive c > 0 e n0 > 0 tali che T(n) >= c F(n) per n > n0
  – *in termini informali T(n) cresce piu’ di F(n) per n grandi*
- Theta grande: diremo che T(n) è Θ(F(n)) se T(n) e’ O(F(n)) e anche Ω(F(n)
  – *in termini informali T(n) cresce come F(n) per N grandi*

Notazione “O grande”

- Nella pratica si utilizzerà informalmente la notazione O grande per indicare che la crescita di T(n) è pari alla crescita di F(n) per n grandi
- Si osservi che la complessità di due algoritmi può essere in entrambi i casi O(F(n)) anche se il secondo è, per esempio, mille volte più veloce del primo
- Pertanto la notazione O grande e’ utile per determinare l’applicabilità di principio di un algoritmo, ma in pratica è necessaria un’analisi più dettagliata
- L’analisi dovra’ anche tenere conto dell’effettiva dimensione pratica del problema. Ad esempio un algoritmo con un andamento migliore per n grandi potrebbe richiedere un tempo inaccettabile per piccoli valori di n
- In un’analisi dettagliata della complessita di dovra’ anche tenere conto del caso medio (Average case), ovvero il tempo medio di esecuzione dell’algoritmo, e del caso peggiore (Worst case), ovvero il tempo richiesto nei casi più “sfortunati”

Ricerca lineare

Ricerca lineare

- Abbiamo già visto un algoritmo da utilizzare per individuare la posizione di un elemento che abbia un particolare valore all'interno di un array i cui elementi ***non siano ordinati***
- Dato che bisogna esaminare tutti gli elementi, si parla di ***ricerca sequenziale*** o ***lineare***

```
public class ArrayAlgorithms
{  public static int linearSearch(int[] a,
      int v)
   {  for (int i = 0; i < a.length; i++)
         if (a[i] == v)
            return i; // TROVATO
      return -1;  // NON TROVATO
   }
   ...
}
```

Ricerca lineare: prestazioni

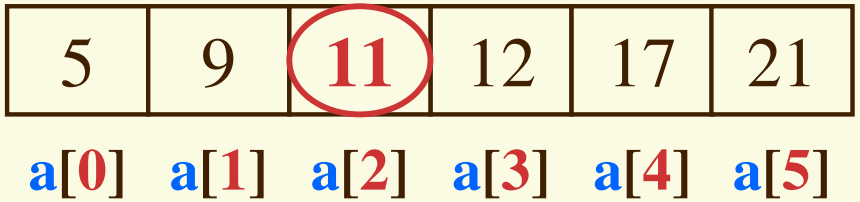
- Valutiamo le prestazioni dell'algoritmo di ricerca lineare
  - se l'elemento cercato non è presente nell'array, sono necessari `n` accessi
  - se l'elemento cercato è presente nell'array, il numero di accessi necessari per trovarlo dipende dalla sua posizione
    - in media saranno **(n+1)/2**
    - **numero di accessi: 1+2+…+n = n(n+1)/2**
    - **media: numero di accessi / n**
- In ogni caso, quindi, le prestazioni dell'algoritmo sono

`O(n)`

Ricerca binaria

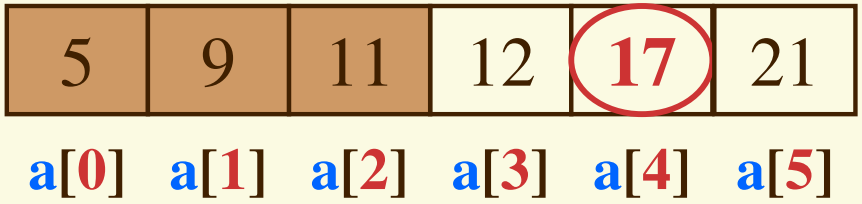
Ricerca in un array ordinato

- Il problema di individuare la posizione di un elemento che abbia un particolare valore all'interno di un array può essere affrontato in modo *più efficiente se l'array è ordinato*
- Cerchiamo l'elemento **12**



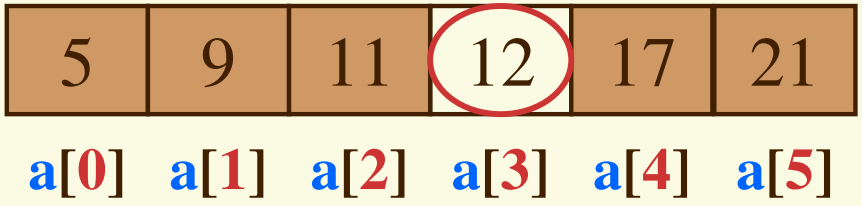
- Confrontiamo **12** con l'elemento che si trova (circa) al centro dell'array, `a[2]`, che è **11**
- L'elemento che cerchiamo è maggiore di **11**
  - *se è presente nell'array, sarà a destra di 11*

Ricerca in un array ordinato



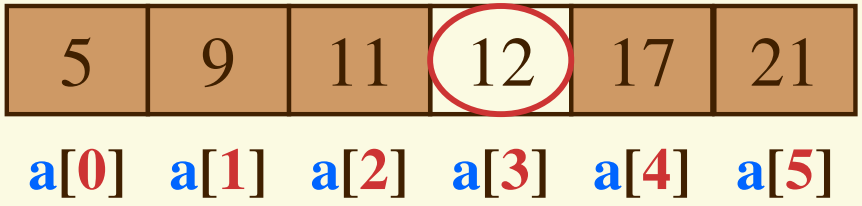
- A questo punto dobbiamo cercare l'elemento **12** nel solo sotto-array che si trova a destra di `a[2]`
- Usiamo lo stesso algoritmo, confrontando **12** con l'elemento che si trova al centro, `a[4]`, che è **17**
- L'elemento che cerchiamo è minore di **17**
  - ***se è presente nell'array, sarà a sinistra di 17***

Ricerca in un array ordinato



- A questo punto dobbiamo cercare l'elemento **12** nel sotto-array composto dal solo elemento `a[3]`
- Usiamo lo stesso algoritmo, confrontando **12** con l'elemento che si trova al centro, `a[3]`, che è **12**
- L'elemento che cerchiamo è uguale a **12**
  - ***l'elemento che cerchiamo è presente nell'array e si trova in posizione 3***

Ricerca in un array ordinato



- Se il confronto tra l'elemento da cercare e l'elemento `a[3]` avesse dato esito negativo, avremmo cercato nel sotto-array ***vuoto*** a sinistra o a destra, concludendo che l'elemento cercato non è presente nell'array
- Questo algoritmo si chiama ***ricerca binaria***, perché a ogni passo si divide l'array ***in due parti***, e funziona soltanto se l'array è ordinato

```
public class ArrayAlgorithms
{  public static int binarySearch(int[] a, int v)
   {  return binSearch(a, 0, a.length - 1, v);
   }
   private static int binSearch(int[] a, int from,
                                int to, int v)
   {  if (from > to) return -1;  // NON TROVATO
      int mid = (from + to) / 2; // CIRCA IN MEZZO
      int middle = a[mid];
      if (middle == v)
         return mid; // TROVATO
      else if (middle < v)
         // CERCA A DESTRA
         return binSearch(a, mid + 1, to, v);
      else
         // CERCA A SINISTRA
         return binSearch(a, from, mid - 1, v);
   }
   ...
}
```

Ricerca binaria: prestazioni

- Valutiamo le prestazioni dell'algoritmo di ricerca binaria in un array ordinato
  - osserviamo che *l'algoritmo è ricorsivo*
- Per cercare in un array di dimensione `n` bisogna
  - effettuare un confronto (con l'elemento centrale)
  - effettuare una ricerca in un array di dimensione `n/2`
- Quindi

```
T(n) = T(n/2) + 1
```

Ricerca binaria: prestazioni

$$T(n) = T(n/2) + 1$$

- La soluzione di questa equazione (equazione fra funzioni) per ottenere $T(n)$ in funzione di n non è semplice e va al di là degli scopi di questo corso
- Si può però agevolmente verificare, per sostituzione, che questa sia la soluzione

$$T(n) = 1 + \log_2 n \quad \Rightarrow \quad O(\log n)$$

- Le prestazioni dell'algoritmo sono quindi ***migliori di quelle della ricerca lineare***

Ricerca binaria iterativa

```
//restituisce l'indice di v nell'array o -1 se
// non trovato
public static int binarySearch(int[] a, int v)
{
   int from = 0;                          //Limite inferiore
   int to = a.length - 1;                 //Limite superiore

//fino a che l'array corrente ha almeno un elemento
   while(from <= to)
   {
      int mid = (from + to)/2;            //CIRCA IN MEZZO

      if (a[mid] == v)
         return mid;                      //TROVATO
      else if (a[mid] < v)
         from = mid + 1;                  //CERCA A DESTRA
      else
         to = mid - 1;                    //CERCA A SINISTRA

   }

   return -1; //NON TROVATO
}
```

Algoritmo O(log n)

Lezione XXIII
Me 14-Nov-2007

Ricerca lineare con sentinella

Quanto tempo serve per trovare un dato fra n?

- Abbiamo analizzato situazioni di questo tipo in altri casi per cui sappiamo subito dire che dobbiamo fare un numero di cicli pari a:
  - *1* nel caso migliore (l'elemento cercato è il primo)
  - *n* nel caso peggiore (l'elemento cercato è l'ultimo o non è presente)
  - *(n+1)/2* in media (l'elemento cercato è presente e occupa con la stessa probabilità tutti le posizioni possibili)
- Ad ogni ciclo dobbiamo fare due confronti, uno per verificare se l'elemento è presente e l'altro per verificare se abbiamo esaurito i dati

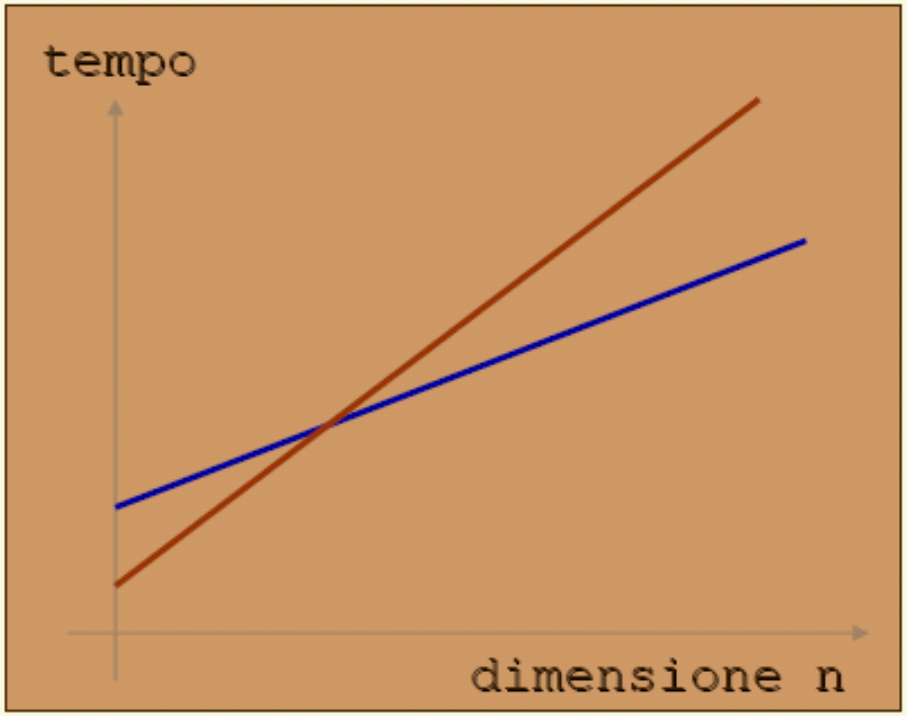
Quanto tempo serve per trovare un dato fra n?

```
int[] v = new int[…];
v[0] = …;
…
int target = …;
for (int i = 0; i < vSize; i++)
  if (target == v[i])
     return i;

return -1;
```

Uso di una sentinella

- È possibile ridurre a metà il numero di confronti utilizzando una ***sentinella*** (utile anche in altre circostanze)
- Supponendo che l'array v[]sia riempito solo in parte e abbia almeno una posizione libera, inseriamo il valore da cercare al posto di indice **vSize**
- A questo punto siamo sicuri che l'elemento cercato compaia nell'array almeno una volta; possiamo quindi interrompere il ciclo di ricerca quando si trova il dato senza controllare se si è esaurito il vettore
- Al termine del ciclo se l'elemento trovato occupa la posizione **vSize** la ricerca è fallita

```
int i;
v[vSize] = target;
for (i = 0; target != v[i]; i++)
    ;
if (i == vSize)
    // non trovato
else
    // trovato
```



Due confronti

Sentinella

Anche usando la sentinella il comportamento asintotico rimane $O(n)$

Cambia la costante di proporzionalità che lega il tempo effettivo a n

Possiamo fare meglio?

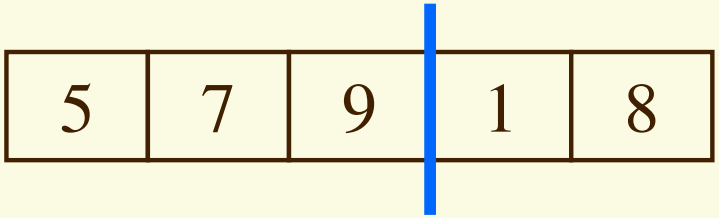
- Se non abbiamo altre informazioni non c'è modo di migliorare le cose.
- Se sappiamo che l'array è ordinato possiamo utilizzare una strategia ricorsiva di tipo *divide et impera* che sfrutta il risultato di ogni confronto per ridurre le dimensioni del problema da risolvere

ricerca binaria

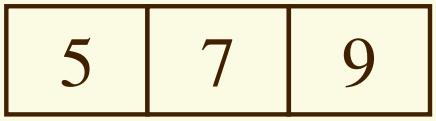
Ordinamento per fusione (*MergeSort*)

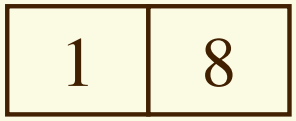
MergeSort (Ordinamento per fusione)

- Presentiamo ora un algoritmo di ordinamento (***MergeSort***) che vedremo avere ***prestazioni migliori*** di quelle dell'ordinamento per selezione
- Dovendo ordinare il vettore seguente



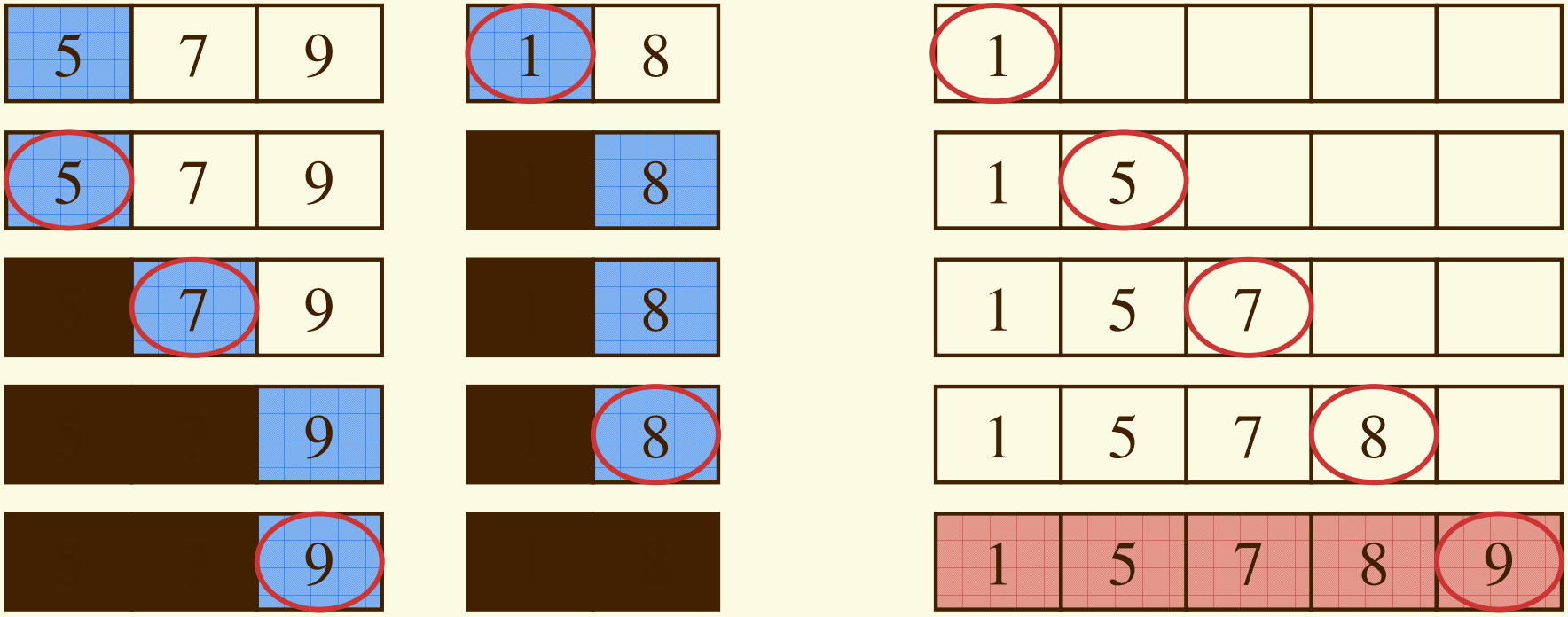
lo dividiamo in due parti (circa) uguali





MergeSort

- Se supponiamo che, come in questo caso, le due parti siano già ordinate, è facile costruire il vettore ordinato, togliendo sempre ***il primo elemento da uno dei due vettori***, scegliendo ***il più piccolo***



MergeSort

- Ovviamente, nel caso generale le due parti del vettore non saranno ordinate
- Possiamo però ***ordinare ciascuna parte ripetendo il processo***
  - dividiamo il vettore in due parti (circa) uguali
  - ***supponiamo*** che siano entrambe ordinate
  - ***uniamo le due parti*** nel modo appena visto
    - questa fase si chiama ***fusione*** (***merge***)
- Continuando così, arriveremo certamente ad una situazione in cui ***le due parti sono davvero ordinate***
  - ***quando contengono un solo elemento!***

MergeSort

- Si delinea quindi un algoritmo ***ricorsivo*** per ordinare un array, chiamato ***MergeSort***
  - se l'array contiene meno di due elementi, è già ordinato
  - altrimenti
    - si divide l'array in due parti (circa) uguali
    - si ordina la prima parte usando ***MergeSort***
    - si ordina la seconda parte usando ***MergeSort***
    - si fondono le due parti ordinate usando l'algoritmo di ***fusione*** (***merge***)

```
public class ArrayAlgorithms
{ public static void mergeSort(int[] a)
  {  // caso base
    if (a.length < 2) return;

    // dividiamo (circa) a meta'
    int mid = a.length / 2;
    int[] left = new int[mid];
    int[] right = new int[a.length - mid];
    System.arraycopy(a, 0, left, 0, mid);
    System.arraycopy(a, mid, right, 0,
       a.length - mid);
    // passi ricorsivi: problemi piu' semplici
    // si tratta di doppia ricorsione
      mergeSort(left);
      mergeSort(right);
      // fusione
      merge(a, left, right);
   }
…
```

```
public class ArrayAlgorithms
{  private static void merge(int[] a, int[] b,
                             int[] c)
   {  int ia = 0, ib = 0, ic = 0;
      //finché ci sono elementi in b e c
      while (ib < b.length && ic < c.length)
         if (b[ib] < c[ic])
            a[ia++] = b[ib++];
         else
            a[ia++] = c[ic++];
      //qui uno dei due array non ha più
      //elementi
      while (ib < b.length)
         a[ia++] = b[ib++];
      while (ic < c.length)
         a[ia++] = c[ic++];
   }
   ...
}
```

Prestazioni di MergeSort

Prestazioni di MergeSort

- Rilevazioni sperimentali delle prestazioni mostrano che l'ordinamento con MergeSort è molto più efficiente dell'ordinamento con selezione
- Cerchiamo di fare una valutazione teorica, calcolando il numero di accessi ai singoli elementi dell'array che sono necessari per l'ordinamento
- Chiamiamo `T(n)` il numero di accessi necessari per ordinare un array di `n` elementi

Prestazioni di MergeSort

- Le due invocazioni di **System.arraycopy** richiedono complessivamente `2n` accessi
  - tutti gli `n` elementi devono essere letti e scritti
- Le due invocazioni ricorsive richiedono `T(n/2)` ciascuna
- La fusione richiede
  - `n` accessi per scrivere gli `n` elementi dell'array ordinato
    - ogni elemento da scrivere richiede la lettura di due elementi, uno da ciascuno dei due array da fondere, per un totale di `2n` accessi

Prestazioni di MergeSort

- Sommando tutti i contributi si ottiene

$$T(n) = 2T(n/2) + 5n$$

- La soluzione di questa equazione per ottenere `T(n)` in funzione di `n` non è semplice e va al di là degli scopi di questo corso
- Si può però agevolmente verificare, per sostituzione, che questa sia la soluzione

$$T(n) = n + 5n*\log_2 n$$

Prestazioni di MergeSort

$$T(n) = n + 5n*\log_2 n$$

$$\begin{aligned}
T(n) &= 2T(n/2) + 5n \\
&= 2[n/2 + 5n/2*\log_2(n/2)] + 5n \\
&= n + 5n*\log_2(n/2) + 5n \\
&= n + 5n*(\log_2 n - \log_2 2) + 5n \\
&= n + 5n*(\log_2 n - 1) + 5n \\
&= n + 5n*\log_2 n - 5n + 5n \\
&= n + 5n*\log_2 n
\end{aligned}$$

Prestazioni di MergeSort

$$T(n) = n + 5n*\log_2 n$$

- Per determinare la notazione "O grande", osserviamo che
  - il termine $5n*\log_2 n$ cresce più rapidamente del termine n
  - il fattore 5 è ininfluente, come tutti i fattori moltiplicativi costanti

$$T(n) = n \log_2 n$$

Prestazioni di MergeSort

$$T(n) = n \log_2 n$$

- Nelle notazioni "O grande" non si indica la base dei logaritmi, perché il logaritmo in una base si può trasformare nel logaritmo in un'altra base con un fattore moltiplicativo, che va ignorato
- Quindi, le prestazioni dell'ordinamento MergeSort hanno un andamento

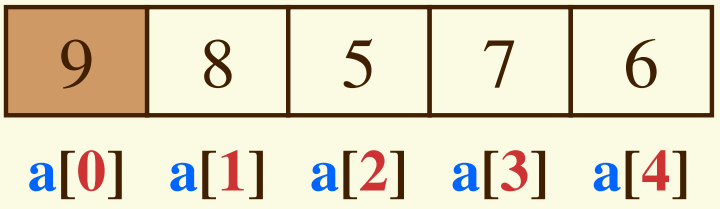
$$O(n \log n)$$

e sono quindi migliori di $O(n^2)$

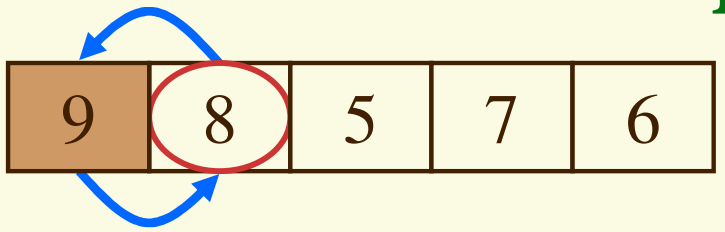
Ordinamento per inserimento

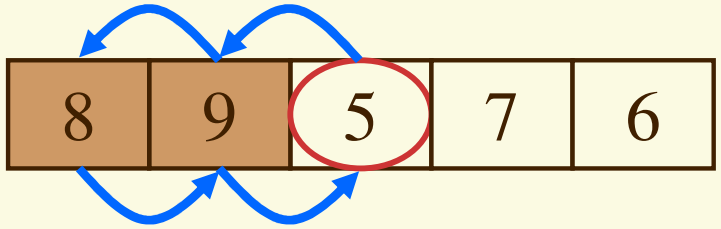
Ordinamento per inserimento

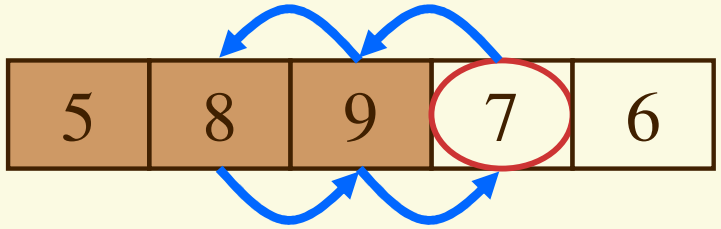
- L'algoritmo di ordinamento per inserimento
  - inizia osservando che il sottoarray di lunghezza unitaria costituito dalla prima cella dell'array è ordinato (essendo di lunghezza unitaria)
  - procede estendendo verso destra la parte ordinata, includendo nel sottoarray ordinato il primo elemento alla sua destra
    - per farlo, il nuovo elemento viene spostato verso sinistra finché non si trova nella sua posizione corretta, spostando verso destra gli elementi intermedi

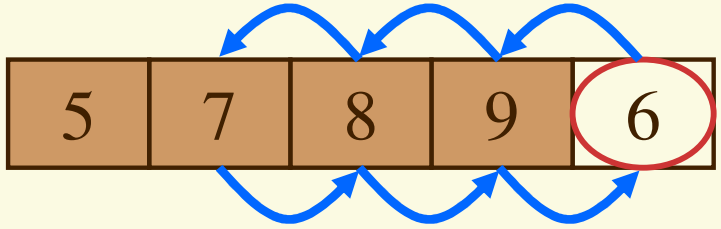


Ordinamento per inserimento









```
public static void insertionSort(int[] a)
{   // il ciclo inizia da 1 perché il primo
    // elemento non richiede attenzione
    for (int i = 1; i < a.length; i++)
    {   // nuovo elemento da inserire
        int temp = a[i];
        // la variabile di ciclo j va definita
        // fuori dal ciclo perché il suo valore
        // finale viene usato in seguito
        int j;
        // vengono spostati a destra di un posto
        // tutti gli elementi a sinistra di temp
        // che sono maggiori di temp, procedendo
        // da destra a sinistra
        for (j = i; j > 0 && temp < a[j-1]; j--)
            a[j] = a[j-1];
        // temp viene inserito in posizione
        a[j] = temp;
    }
}
```

Analisi teorica delle prestazioni

- Ordiniamo con inserimento un array di `n` elementi
- Il ciclo esterno esegue `n-1` iterazioni
  - a ogni iterazione vengono eseguiti
    - 2 accessi (uno prima del ciclo interno ed uno dopo)
    - il ciclo interno
- Il ciclo interno esegue 3 accessi per ogni sua iterazione
  - ma quante iterazioni esegue?
    - dipende da come sono ordinati i dati!

Ordinamento per inserimento

- **Caso migliore**: i dati sono già ordinati
- Il ciclo più interno non esegue mai iterazioni
  - richiede un solo accesso per la prima verifica
- Il ciclo esterno esegue `n-1` iterazioni
  - ad ogni iterazione vengono eseguiti
    - 2 accessi (uno prima del ciclo interno ed uno dopo)
    - 1 accesso (per verificare la condizione di terminazione del ciclo interno)
- `T(n) = 3*(n-1) = O(n)`

Ordinamento per inserimento

- **Caso peggiore**: i dati sono ordinati a rovescio
- Ciascun nuovo elemento inserito richiede lo spostamento di tutti gli elementi alla sua sinistra, perché deve essere inserito in posizione 0
- $T(n) = (2 + 3*1) + (2 + 3*2) + (2 + 3*3) + \ldots + (2 + 3*(n-1))$
  $= 2(n-1) + 3[1+2+\ldots+(n-1)]$
  $= 2(n-1) + 3n(n-1)/2$
  $= O(n^2)$

Ordinamento per inserimento

- **Caso medio**: i dati sono in ordine casuale
- Ciascun nuovo elemento inserito richiede in media lo spostamento di metà degli elementi alla sua sinistra
- $T(n) = (2 + 3*1/2) + (2 + 3*2/2) + (2 + 3*3/2) + \ldots + (2 + 3*(n-1)/2)$
  $= 2(n-1) + 3[1+2+\ldots+(n-1)]/2$
  $= 2(n-1) + 3[n(n-1)/2]/2$
  $= O(n^2)$

Confronto tra ordinamenti

	caso migliore	caso medio	caso peggiore
merge sort	$n \lg n$	$n \lg n$	$n \lg n$
selection sort	n^2	n^2	n^2
insertion sort	n	n^2	n^2

- Se si sa che l'array è "quasi" ordinato, è meglio usare l'ordinamento per inserimento
- Esempio notevole: un array che viene mantenuto ordinato per effettuare ricerche, inserendo ogni tanto un nuovo elemento e poi riordinando

Qualche numero per confrontare

$T_1(n) = n^2$ con $T_2(n) = n \log n$

n	n * n	n log n	n * n / n log n
1,00E+01	1,00E+02	6,64E+01	1,51E+00
1,00E+02	1,00E+04	1,33E+03	7,53E+00
1,00E+03	1,00E+06	1,99E+04	5,02E+01
1,00E+04	1,00E+08	2,66E+05	3,76E+02
1,00E+05	1,00E+10	3,32E+06	3,01E+03
1,00E+06	1,00E+12	3,99E+07	2,51E+04
1,00E+07	1,00E+14	4,65E+08	2,15E+05

Lezione XXIV
Gi 15-Nov-2007

Numeri Casuali

Numeri casuali

- Come facciamo a generare un numero casuale al calcolatore?
  - sappiamo che il calcolatore fa sempre esattamente ciò che gli viene detto di fare mediante un programma, si comporta cioè in modo ***deterministico***, e non casuale!
- Esistono complesse teorie matematiche che forniscono algoritmi in grado di generare sequenze di numeri ***pseudo-casuali*** (cioè ***quasi casuali***), che sono un'ottima approssimazione di sequenze di numeri casuali

Numeri casuali

- La classe **Math** mette a disposizione il metodo **random( )** per generare sequenze di numeri pseudo-casuali
- Ogni invocazione del metodo restituisce un numero reale pseudo-casuale nell'intervallo **[0, 1[**

```
double x = Math.random();
```

- Per ottenere, ad esempio, numeri interi casuali compresi nell'intervallo **[a, b]**, basta fare un po' di calcoli…

```
int n = (int)(a + (1+b-a)*Math.random());
```

Esempio

```
public class AverageDice
{  public static void main(String[] args)
   {  final int TRIALS = 100000;
      int sum = 0;
      for (int i = 0; i < TRIALS; i++)
      {  // nuovo lancio del dado
         int d = (int)(1 + 6 * Math.random());
         sum = sum + d;
      }
      double avg = ((double)sum) / TRIALS;
      System.out.println("Average: " + avg);
   }
}
```

Numeri casuali

- Esiste anche una classe della libreria standard
  - java.util.Random

```
import java.util.Random
public class UnaClasse
{
   public void unMetodo()
   {  //se servono numeri casuali
      Random rand = new Random();
      int n = rand.nextInt();
      double d = rand.nextDouble();//[0,1[
      long l = rand.nextLong();
…
}
```

Numeri casuali

- Nella classe java.util.Random il metodo nextInt() è sovraccarico, come segue:
  - `int nextInt()` restituisce un numero pseudo-casuale intero nell'intervallo [`Integer.MIN_VALUE`, `Integer.MAX_VALUE`].
  - `int nextInt(int k)` restituisce un numero pseudo-casuale intero nell'intervallo `[0, k[`.

Numeri casuali

```
import java.util.Random;
public class AverageDice2
{  public static void main(String[] args)
   {  final int TRIALS = 100000;
      Random rand = new Random();
      int sum = 0;
      for (int i = 0; i < TRIALS; i++)
      {  // nuovo lancio del dado
         int d = 1 + rand.nextInt(6);
         sum = sum + d;
      }
      double avg = ((double)sum) / TRIALS;
      System.out.println("Average: " + avg);
   }
}
```

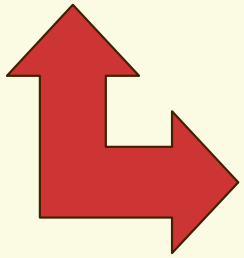
Esempio

- Eseguendo il programma **AverageDice** più volte, si ottiene un diverso valore medio ogni volta, a conferma che i numeri casuali generati variano ad ogni esecuzione, come dev'essere
- Si osserva che il valore medio che si ottiene è sempre ***piuttosto vicino*** a 3.5
- La "teoria dei grandi numeri" afferma che il valore 3.5 viene raggiunto esattamente con un numero infinito di lanci del dado
  - dal punto di vista pratico, è sufficiente un numero di lanci di qualche milione…(ma e' vero?)

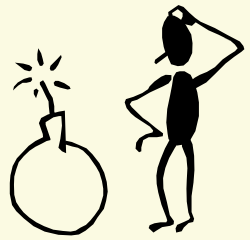
Espressioni Canoniche (Regular Expressions) cenni

- Supponiamo di voler risolvere il seguente problema:
  - acquisire i nomi di alcune citta' contenuti in una stringa e separati dal carattere /
  - Esempio: "Buenos Aires/Los Angeles/La Paz"

```
String line = "Buenos Aires/Los Angeles/La Paz";
Scanner st = new Scanner(line);
while (st.hasNext())
   System.out.println(st.next());
// Non Funziona
```



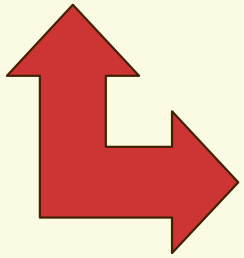
```
Buenos
Aires/Los
Angeles/La
Paz
```



Espressioni Canoniche (Regular Expressions)

- Il frammento di codice non funziona perchè l'oggetto st di classe Scanner assume come separatori di default l'insieme dei caratteri *whitespaces* che non contiene il carattere '/'

```
String line = "Buenos Aires/Los Angeles/La Paz";
Scanner st = new Scanner(line);
st.useDelimiter("[/]+");
while (st.hasNext())
    System.out.println(st.next());
// Funziona
```



```
Buenos Aires
Los Angeles
La Paz
```

Espressioni Canoniche (Regular Expressions)

- Il metodo **`useDelimiter(String pattern)`** della classe Scanner permette di impostare un nuovo insieme di caratteri delimitatori
- Il metodo assume come parametro una stringa contenente **un'espressione canonica (regular expression)**
- I nuovi caratteri delimitatori vengono inseriti in una coppia di parentesi quadre
  - **`[abc]`** significa il carattere **`'a'`** o il carattere **`' b '`** o il carattere **`' c '`**
- La coppia di parentesi quadra e' seguita dal carattere + che ha il significato una o piu' volte (i caratteri delimitatori possono essere ripetuti piu' volte)
- Esempio: se si vuole che l'insieme dei caratteri delimitatori contenga i caratteri **`':'`** **`'@'`** **`'!'`** si scrive

```
...
Scanner st = new Scanner(line);
st.useDelimiter("[:@!]+");
```

Espressioni Canoniche (Regular Expressions)

- Le espressioni canoniche sono usate in parecchi programmi di servizio come il comando **grep** in Linux
- Ad esempio per elencare nel file *programma.java* tutte le righe che contengono dei numeri si puo inviare il comando

```
$grep [0-9]+ programma.java
```

- **0-9** significa l'insieme di caratteri da 0 a 9 (intervallo)
- Se vogliamo escludere tutte le stringhe in cui i numeri sono preceduti da caratteri alfabetici possiamo scrivere

```
$grep [^A-Za-z][0-9]+ programma.java
```

- **^A-Za-z** significa qualsiasi carattere (^) ad eccezione dei caratteri nell'insieme da 'A' a ' Z ' e da ' a ' a ' z '