Lezione XXI
Me 2-Nov-2005

Algoritmi di 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) memorizzato 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

11	9	17	5	12
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

5	9	11	12	17
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

Ordinamento per selezione

11	9	17	5	12
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

5	9	17	11	12

- 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

5	9	17	11	12
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

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***

5	9	17	11	12
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

Ordinamento per selezione

5	9	17	11	12
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

- 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]**

5	9	11	17	12

5	9	11	17	12
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

Ordinamento per selezione

5	9	11	17	12
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

- 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]**

5	9	11	12	17

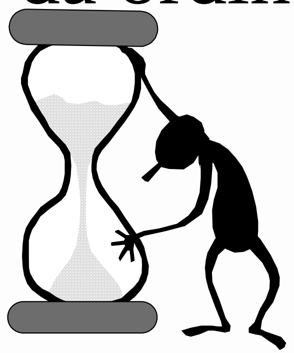
5	9	11	12	17
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

- 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

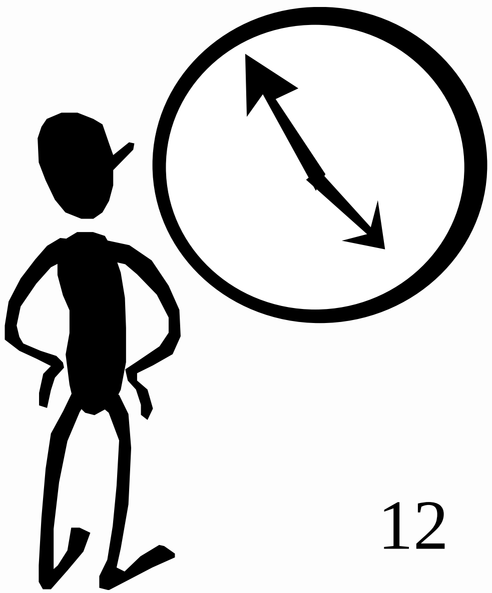
- L'algoritmo di ordinamento per selezione è corretto ed è in grado di ordinare qualsiasi array
  - allora perché studieremo altri algoritmi di ordinamento?
  - ***a cosa serve avere diversi algoritmi per risolvere lo stesso problema?***
- Vedremo che ***esistono algoritmi di ordinamento che***, a parità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

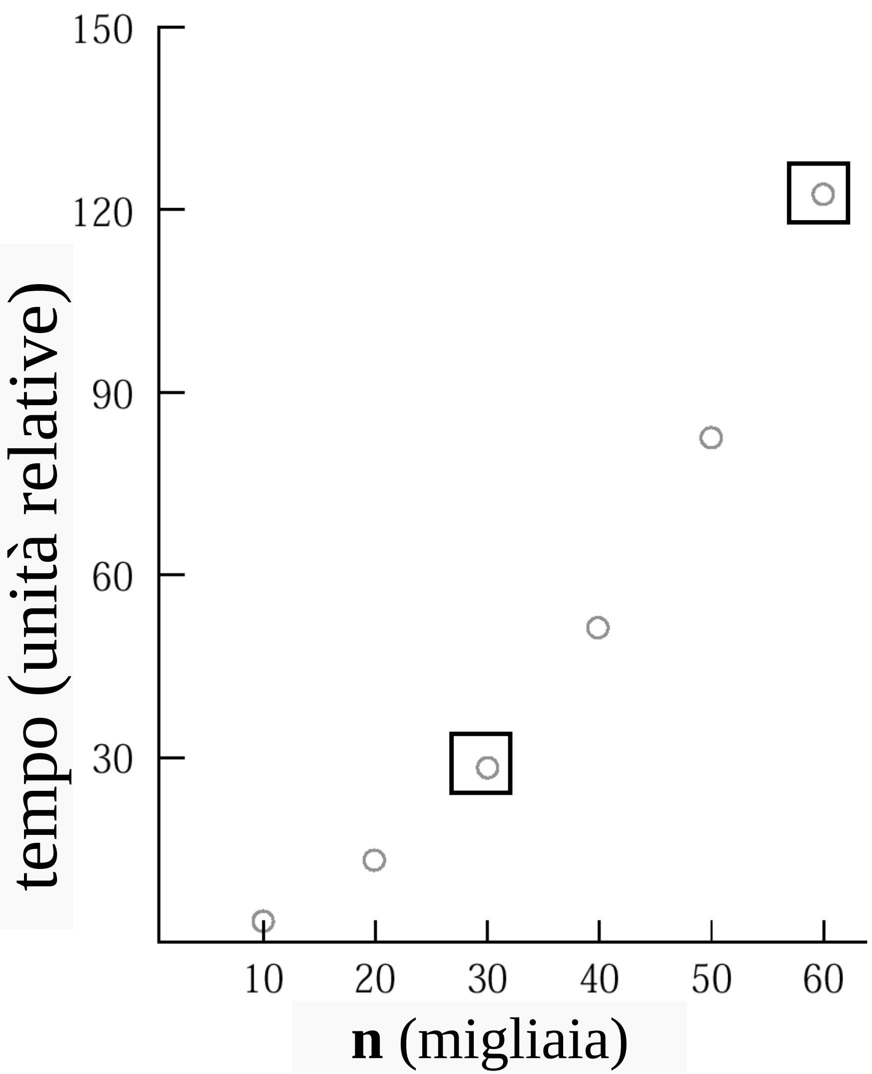
- Il tempo di esecuzione di un algoritmo va misurato all’interno del programma
- Si usa il metodo statico

  `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)+…+(3+4)+(2+4)
  = n+(n-1)+…+3+2 + (n-1)*4
  = n*(n+1)/2 - 1 + (n-1)*4
  = 1/2*n² + 9/2*n - 5
```

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

```
n+(n-1)+…+3+2+1 = n*(n+1)/2
```

Andamento asintotico delle prestazioni

$$\boxed{1/2*n^2 + 9/2*n - 5}$$

- 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

$\boxed{\mathtt{1/2*n^2}}$

- Facciamo un'ulteriore semplificazione
  - 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***

$$T(n) = 1/2*n^2$$

$$T(2n) = 1/2*(2n)^2 = 1/2*4*n^2$$

$$= 4*T(n)$$

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

  anche per $T(n) = n^2$

  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 e n_0 tali che T(n) <= c F(n) per n > n_0
  – *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 e n_0 tali che T(n) >= c F(n) per n > n_0
  – *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); cioe’ esistono due costanti c1 e c2 positive e un intero positivo n_0 tali che c1 F(n) <= T(n) <= c2 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 target)
   {  for (int i = 0; i < a.length; i++)
         if (a[i] == target)
            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

$$\boxed{O(n)}$$

Ricerca lineare con e senza sentinella

Quanto tempo occorre 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 <u>due confronti</u>, uno per verificare se l'elemento è presente e l'altro per verificare se abbiamo esaurito i dati

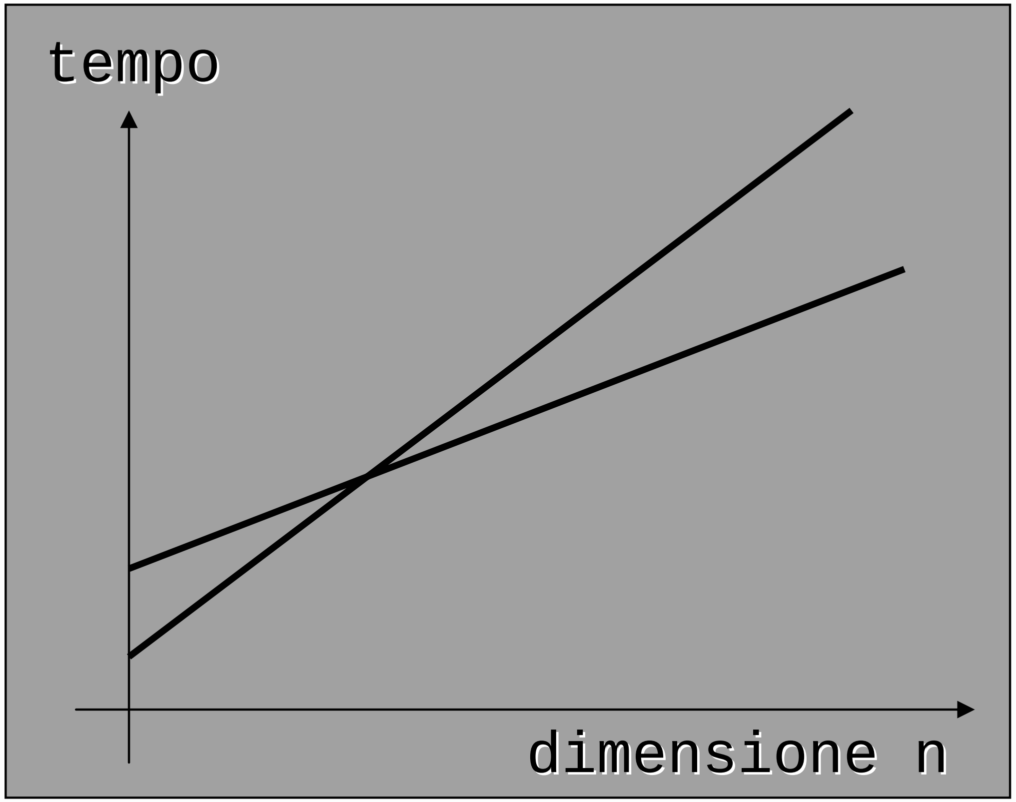
Quanto tempo occorre per trovare un dato fra n?

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

L’uso di una *sentinella*

- È possibile ridurre a metà il numero di confronti utilizzando una *sentinella* (utile anche in altre circostanze)
- Supponendo che il vettore `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 nel vettore 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 <u>l'array è ordinato</u> 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

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 <u>l’array è ordinato</u>***
- Cerchiamo l’elemento **12**

5	9	**11**	12	17	21
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**	**a[5]**

- 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

5	9	11	12	**17**	21
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**	**a[5]**

- 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

5	9	11	12	17	21
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**	**a[5]**

- 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

5	9	11	12	17	21
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**	**a[5]**

- 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**

```java
public class ArrayAlgorithms
{  public static int binarySearch(int[] a, int v)
   {  return binSearch(a, 0, a.length - 1, v);
   }
   private static int binSearch(int[] b, int from,
                                    int to, int target)
   {  if (from > to) return -1;  // NON TROVATO

      int mid = (from + to) / 2; // CIRCA IN MEZZO
      int middle = b[mid];
      if (middle == target)
         return mid; // TROVATO

      else if (middle < target)
         // CERCA A DESTRA
         return binSearch(b, mid + 1, to, target);
      else
         // CERCA A SINISTRA
         return binSearch(b, from, mid - 1, target);
   }
   ...
}
```

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

$$\boxed{\mathtt{T(n) = T(n/2) + 1}}$$

Ricerca binaria: prestazioni

$$\boxed{T(n) = T(n/2) + 1}$$

- 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

$$T(n) = 1 + \log_2 n \Rightarrow \boxed{O(\log n)}$$

- 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 low = 0;//Limite inferiore array corrente
  int high = a.length – 1;//Limite superiore
  int mid;
//fino a che l’array corrente ha almeno un elemento
  while(low <= high)
  { mid = (low + high)/2; //CIRCA IN MEZZO
    if (a[mid] == v) return mid; //TROVATO
    //CERCA A DESTRA
    else if (a[mid] < v) low = mid + 1;
    //CERCA A SINISTRA
    else high = mid - 1;
 }
  return –1; //NON TROVATO
}
```

Algoritmo O(log n)

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();
…
}
```

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?)

Lezione XXII
Gi 3-Nov-2005

Algoritmi di Ordinamento
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

5	7	9	1	8

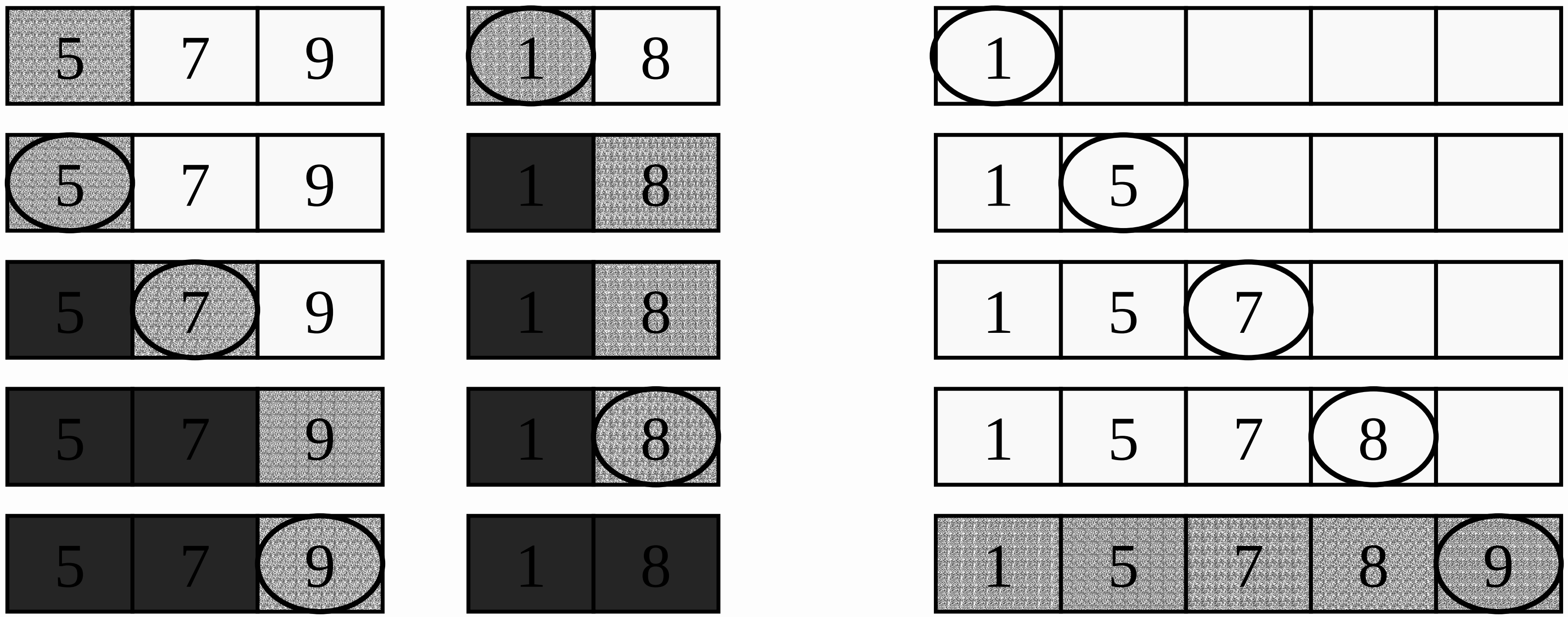
lo dividiamo in due parti (circa) uguali

5	7	9

1	8

MergeSort

- Se supponiamo che, come in questo caso, le due parti siano **già ordinate**, è facile costruire l'array 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 (caso base)
  - Altrimenti (passi ricorsivi)
    - 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

$$\boxed{\texttt{T(n) = n + 5n*log}_2\texttt{n}}$$

- 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

$$\boxed{\texttt{T(n) = n log}_2\texttt{n}}$$

Prestazioni di MergeSort

$$\boxed{\mathbf{T(n) = n\ log_2 n}}$$

- 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

$$\boxed{\mathbf{O(n\ log\ n)}}$$

e sono quindi migliori di $\mathbf{O(n^2)}$

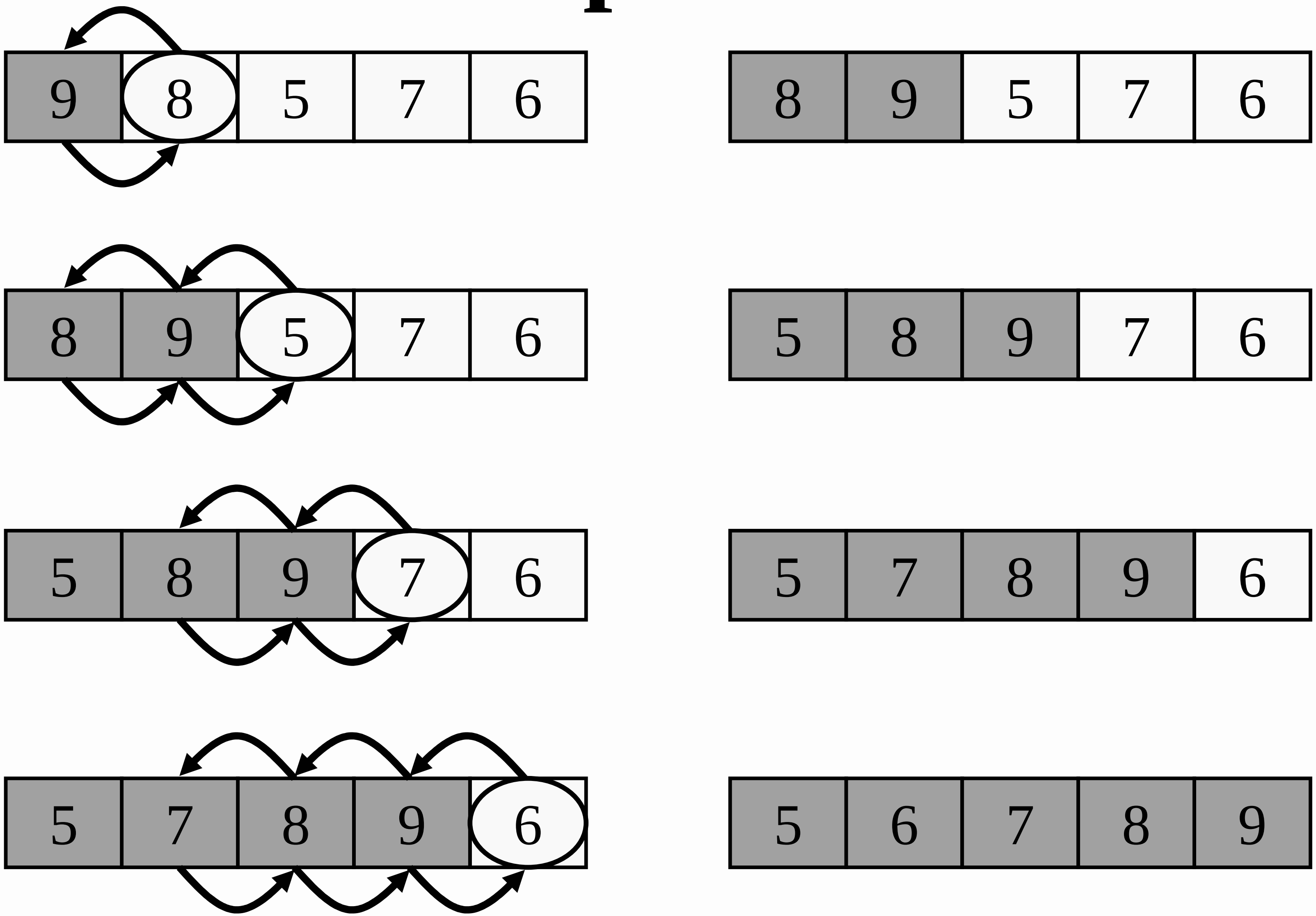
Ordinamento per inserimento

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

9	8	5	7	6
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

Ordinamento per inserimento



```java
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) + ... +
       (2 + 3*(n-1))
     = 2(n-1) + 3[1+2+...+(n-1)]
     = 2(n-1) + 3n(n-1)/2
     = O(n²)
```

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) +`**

```
T(n) = (2 + 3*1/2) + (2 + 3*2/2) +
       (2 + 3*3/2) + ... +
       (2 + 3*(n-1)/2)
     = 2(n-1) + 3[1+2+...+(n-1)]/2
     = 2(n-1) + 3[n(n-1)/2]/2
     = O(n²)
```

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

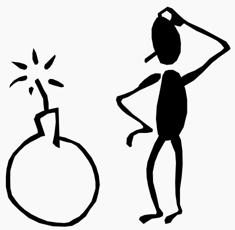
Qualche numero per confrontare $T_1(n) = n^2$ con $T_2(n) = n \log n$

n	n^2	n log n	n^2 / 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

Espressioni Canoniche (Regular Expression) 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 Expression)

- 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 '

Esercitazione

- Esempio: si scriva la classe ArchivioStudenti che memorizza i dati di un insieme di studenti
- Ciascun studente sia identificato da
  - nome
  - numero di matricola
- I metodi pubblici della classe siano:
  - **`public ArchivioStudenti(String filename)`** // costruttore
  - **`public void aggiungi(String n, int m)`** // aggiunge uno studente
  - **`public Studente togli()`** //estrae il primo studente dall’archivio
  - **`public boolean vuoto()`** // restituisce true se l’archivio e’ vuoto
- Il costruttore della classe acquisisca i dati da un file di testo. Nel file i dati di ciascun studente siano descritti in una riga con il formato:
  - nome cognome:::matricola
- Si renda la classe eseguibile, stampando gli studenti dell’archivio in un file. I file di ingresso e uscita siano passati come parametri sulla riga di comando.

Esercitazione

```
class Studente
{
  private String nome;
  private int matricola;

  public Studente(String unNome, int unaMatr)
  {
    nome = unNome;
    matricola = unaMatr;
  }

  public String toString() {return “nome: “
             + nome + " matr.: " + matricola;}
}
```

nome: Marco Rossi matr.: 12345

Esercitazione

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;

import java.util.Scanner;
import java.util.NoSuchElementException;

public class ArchivioStudenti
{
  Studente[] lista;
  int listaSize;

  public ArchivioStudenti()
  {
    lista = new Studente[1];
    listaSize = 0;
  }
  …
```

Esercitazione

```
public ArchivioStudenti(String filename)
  throws FileNotFoundException, IOException
{
  this();

  Scanner in = new Scanner(new FileReader(filename));

  while (in.hasNextLine())
  {
    String line = in.nextLine();

    Scanner tok = new Scanner(line);
    tok.useDelimiter("[:]+");
    String nome = tok.next();
    int matr = tok.nextInt();

    aggiungi(nome, matr);
  }
  in.close();
}
…
```

```
FileReader r = new FileReader(filename);
Scanner in = new Scanner(r);
```

Esercitazione

```
public boolean vuoto()
{
  return listaSize == 0;
}

/**
  aggiunge uno studente all’archivio
  @param unNome nome dello studente
  @param unaMatr matricola dello studente
*/
public void aggiungi(String unNome, int unaMatr)
{
  if (pieno())
    lista = resize(lista, 2 * lista.length);

  lista[listaSize] = new Studente(unNome, unaMatr);
  listaSize++;
}
…
```

Esercitazione

```
/**
   restituisce il primo studente dell’archivio
   cancellandolo
   @return lo studente cancellato
   @throws java.util.NoSuchElementException
*/
public Studente togli()
{
  if (vuoto())
     throw new NoSuchElementException();

  Studente tmpStudent = lista[0];
  for (int i = 0; i < listaSize -1; i++)
    lista[i] = lista[i + 1];

  lista[listaSize - 1] = null; //garbage collector
  listaSize--;

  return tmpStudent;
}
…
```

Esercitazione

```
public static void main(String[] args)
  throws IOException
{
  if (args.length < 2)
  {
    System.out.println("uso: $java”
        + “ ArchivioStudenti file1 file2");
    return;
  }

  ArchivioStudenti ar = new ArchivioStudenti(args[0]);
  PrintWriter out = new PrintWriter(args[1]);

  while (!ar.vuoto())
    out.println(ar.togli());

  out.close();
}
```

Esercitazione

```
  private boolean pieno()
  {
    return listaSize == lista.length;
  }

  private static Studente[] resize(Studente[] oldArray,
                                   int newLength)
  {
     int minLength = oldArray.length < newLength ?
                       oldArray.length : newLength;
     Studente[] newArray = new Studente[newLength];
     for (int i = 0; i < minLength; i++)
       newArray[i] = oldArray[i];

     return newArray;
  }
} // fine della classe ArchivioStudenti
```