Capitolo 13
Ordinamento e ricerca

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Obiettivi del capitolo

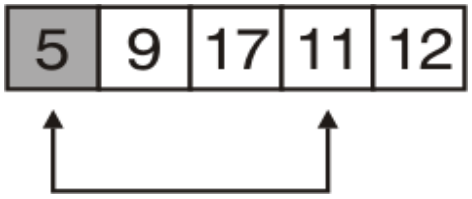
- Studiare alcuni algoritmi di ordinamento e ricerca
- Osservare che algoritmi che risolvono lo stesso problema possono avere prestazioni molto diverse
- Capire la notazione O-grande
- Imparare a stimare le prestazioni di algoritmi e a confrontarle
- Imparare a misurare il tempo d'esecuzione di un programma

Ordinamento per selezione

- L'algoritmo di ordinamento per selezione ordina un array cercando ripetutamente l'elemento minore della regione terminale non ancora ordinata e spostando tale elemento all'inizio della regione stessa.
- Esempio: consideriamo l'ordinamento di un array di numeri interi:

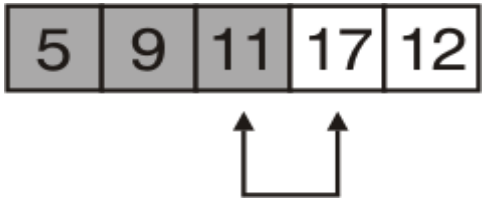
Ordinamento di un array di numeri interi

- Trovare l'elemento più piccolo e spostarlo all'inizio



- Prendere il più piccolo degli elementi rimanenti. Si trova già nella posizione giusta

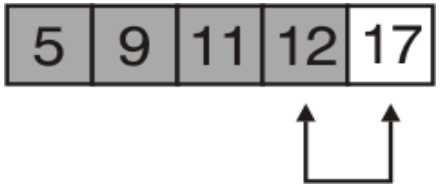
- Ripetiamo il processo



Continua…

Ordinamento di un array di numeri interi

- Ripetere



- Questo ci lascia con una regione non ancora elaborata di lunghezza 1, ma naturalmente una regione di lunghezza 1 è sempre ordinata. Abbiamo finito.

File SelectionSorter.java

```
01: /**
02:    Questa classe ordina un array,
03:    usando l'algoritmo di ordinamento per selezione.
04: */
05: public class SelectionSorter
06: {
07:    /**
08:       Costruisce un ordinatore per selezione.
09:       @param anArray l'array da ordinare
10:    */
11:    public SelectionSorter(int[] anArray)
12:    {
13:       a = anArray;
14:    }
15:
16:    /**
17:       Ordina l'array gestito da questo ordinatore.
18:    */
```

Continua…

File SelectionSorter.java

```
19:    public void sort()
20:    {
21:       for (int i = 0; i < a.length - 1; i++)
22:       {
23:          int minPos = minimumPosition(i);
24:          swap(minPos, i);
25:       }
26:    }
27:
28:    /**
29:       Trova il minimo in una parte terminale     dell'array.
30:       @param from la prima posizione in a che va considerata
31:       @return la posizione dell'elemento minimo presente
32:               nell'intervallo a[from]...a[a.length - 1]
33:    */
34:    private int minimumPosition(int from)
35:    {
```

Continua…

File SelectionSorter.java

```
36:          int minPos = from;
37:          for (int i = from + 1; i < a.length; i++)
38:             if (a[i] < a[minPos]) minPos = i;
39:          return minPos;
40:       }
41:
42:       /**
43:          Scambia due elementi nell'array.
44:          @param i la posizione del primo elemento
45:          @param j la posizione del secondo elemento
46:       */
47:       private void swap(int i, int j)
48:       {
49:          int temp = a[i];
50:          a[i] = a[j];
51:          a[j] = temp;
52:       }
```

Continua…

File SelectionSorter.java

```
53:
54:    private int[] a;
55: }
```

File SelectionSortDemo.java

```
01: import java.util.Arrays;
02: /** Questo programma applica l'algoritmo di ordinamento
03:    per selezione a un array contenente numeri casuali.
04: */
05: public class SelectionSortDemo
06: {
07:    public static void main(String[] args)
08:    {
09:       int[] a = ArrayUtil.randomIntArray(20, 100);
10:       ArrayUtil.print(a);
11:
12:       SelectionSorter sorter = new SelectionSorter(a);
13:       sorter.sort();
14:
15:       System.out.println(Arrays.toString(a));
16:    }
17: }
18:
19:
```

File ArrayUtil.java

```
01: import java.util.Random;
02:
03: /**
04:    Questa classe contiene metodi utili per
05:    la manipolazione di array.
06: */
07: public class ArrayUtil
08: {
09:    /**
10:       Costruisce un array contenente valori casuali.
11:       @param length la lunghezza dell'array
12:       @param n il numero di valori casuali possibili
13:       @return un array contenente length numeri
14:               casuali compresi fra 0 e n - 1
15:    */
16:    public static int[] randomIntArray(int length, int n)
17:    {
```

Continua…

File ArrayUtil.java

```
18:          int[] a = new int[length];
19:          for (int i = 0; i < a.length; i++)
20:             a[i] = generator.nextInt(n);
21:
22:          return a;
23:       }
24:
25:       private static Random generator = new Random();
26: }
```

Continua…

File `ArrayUtil.java`

- Visualizza

```
65 46 14 52 38 2 96 39 14 33 13 4 24 99 89 77 73 87 36 81
2 4 13 14 14 24 33 36 38 39 46 52 65 73 77 81 87 89 96 99
```

Misurazione delle prestazioni dell'algoritmo di ordinamento per selezione

- Vogliamo misurare quanto tempo si impiega per eseguire un algoritmo:

  - escludere il tempo per caricare il programma dal disco alla memoria

  - escludere il tempo per visualizzare i risultati sullo schermo
- Utilizzeremo una classe `StopWatch`, che funziona proprio come un cronometro vero:

  - potete farlo partire, fermarlo e leggere il tempo trascorso

  - usa il metodo `System.currentTimeMillis`
- Creare uno `StopWatch`

  - potete farlo partire, fermarlo e leggere il tempo trascorso

File StopWatch.java

```
01: /**
02:    Un cronometro accumula il tempo mentre è in azione.
03:    Potete avviare e arrestare ripetutamente il cronometro.
04:    Potete utilizzare un cronometro per misurare il tempo
       di esecuzione di un programma.
05: */
06: public class StopWatch
07: {
08:    /**
09:       Costruisce un cronometro fermo e senza tempo accumulato.
11:    */
12:    public StopWatch()
13:    {
14:       reset();
15:    }
16:
```

Continua…

File StopWatch.java

```
17:     /**
18:        Fa partire il cronometro, iniziando ad accumulare il tempo.
19:     */
20:     public void start()
21:     {
22:        if (isRunning) return;
23:        isRunning = true;
24:        startTime = System.currentTimeMillis();
25:     }
26:
27:     /**
28:        Ferma il cronometro. Il tempo non viene più accumulato e il
   tempo trascorso viene sommato al tempo totale.
30:     */
31:     public void stop()
32:     {
```

Continua…

File `StopWatch.java`

```
33:         if (!isRunning) return;
34:         isRunning = false;
35:         long endTime = System.currentTimeMillis();
36:         elapsedTime = elapsedTime + endTime - startTime;
37:      }
38:
39:      /**
40:         Restituisce il tempo totale trascorso.
41:         @return il tempo totale trascorso
42:      */
43:      public long getElapsedTime()
44:      {
45:         if (isRunning)
46:         {
47:            long endTime = System.currentTimeMillis();
48:            return elapsedTime + endTime - startTime;
49:         }
```

Continua…

File StopWatch.java

```
50:          else
51:             return elapsedTime;
52:       }
53:
54:       /**
55:       Ferma il cronometro e azzera il tempo totale trascorso.
56:       */
57:       public void reset()
58:       {
59:          elapsedTime = 0;
60:          isRunning = false;
61:       }
62:
63:       private long elapsedTime;
64:       private long startTime;
65:       private boolean isRunning;
66: }
```

File SelectionSortTimer.java

```
01: import java.util.Scanner;
02:
03: /**
04:    Questo programma misura il tempo richiesto
05:    per ordinare con l'algoritmo di ordinamento
06:    per selezione un array di dimensione
       specificata dall'utente.
07: */
08: public class SelectionSortTimer
09: {
10:    public static void main(String[] args)
11:    {
12:       Scanner in = new Scanner(System.in);
13:       System.out.print("Enter array size: ");
14:       int n = in.nextInt();
15:
16:       // costruisce un array casuale
17:
```

Continua…

File SelectionSortTimer.java

```
18:         int[] a = ArrayUtil.randomIntArray(n, 100);
19:         SelectionSorter sorter = new SelectionSorter(a);
20:
21:         // usa il cronometro per misurare il tempo
22:
23:         StopWatch timer = new StopWatch();
24:
25:         timer.start();
26:         sorter.sort();
27:         timer.stop();
28:
29:         System.out.println("Elapsed time: "
30:               + timer.getElapsedTime() + " milliseconds");
31:     }
32: }
33:
34:
```

File `SelectionSortTimer.java`

- Visualizza

```
Enter array size: 100000
Elapsed time: 27880 milliseconds
```

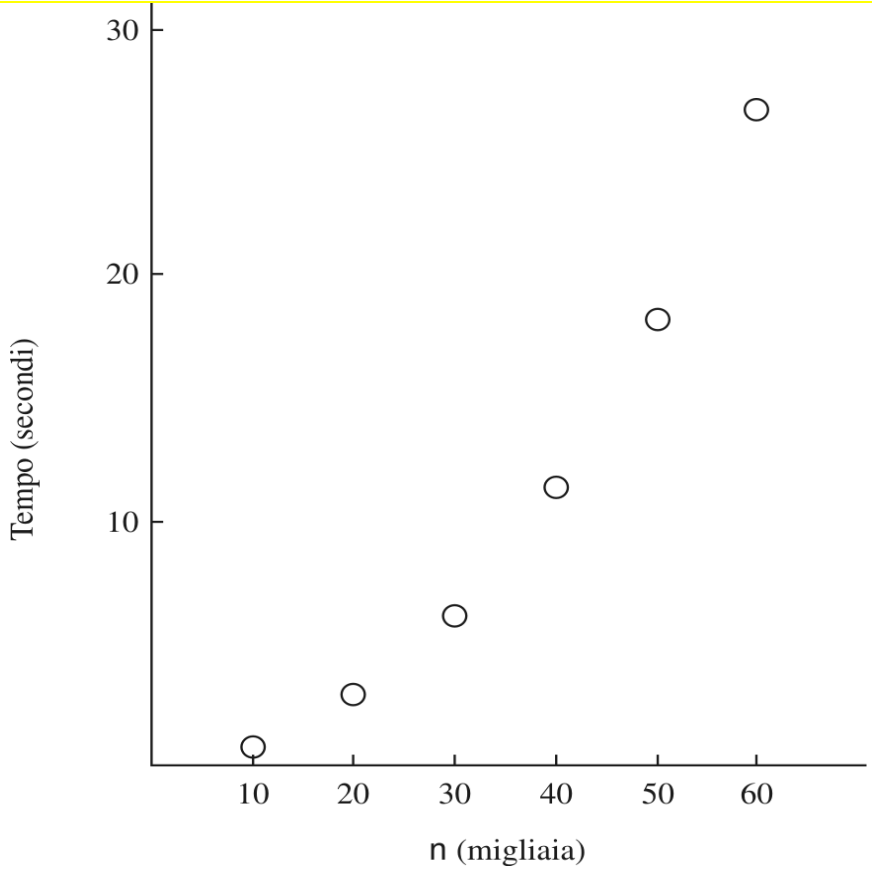
Tempo impiegato dall'ordinamento per selezione

n	Millisecondi
10 000	772
20 000	3051
30 000	6846
40 000	12 188
50 000	19 015
60 000	27 359

Rilevazioni ottenute con un processore Pentium a 1.2 GHz con sistema operativo Linux e Java 5.0

Tempo impiegato dall'ordinamento per selezione

Figura 1



Raddoppiando le dimensioni dell'insieme dei dati, il tempo che occorre per ordinarli è più del doppio.

Analisi delle prestazioni dell'algoritmo di ordinamento per selezione

- Supponiamo che *n* sia la dimensione dell'array.

  - dobbiamo trovare il più piccolo fra *n* numeri, poi scambiamo gli elementi, cosa che richiede due visite

  - dobbiamo visitare soltanto *n* –1 elementi per trovare il minimo.

  - vengono visitati soltanto *n* – 2 elementi, e l'ultimo passo visita soltanto due elementi tra i quali deve trovare il minimo.

Analisi delle prestazioni dell'algoritmo di ordinamento per selezione

- Il numero totale delle visite è:

  - $n + 2 + (n - 1) + 2 + (n - 2) + 2 + \ldots + 2 + 2 = n + (n - 1) + \ldots + 2 + (n - 1) \cdot 2$

  $= 2 + \ldots + (n - 1) + n + (n - 1) \cdot 2$

  $$= \frac{n \cdot (n+1)}{2} - 1 + (n-1) \cdot 2$$

  perché $$1 + 2 + \ldots + (n-1) + n = \frac{n \cdot (n+1)}{2}$$

  - si semplifica in $-n \quad + -n -$

  - ignorare 5/2 · n - 3

  - ignoriamo anche il fattore costante 1/2: non ci interessa il conteggio effettivo delle visite per un singolo valore di *n*

Analisi delle prestazioni dell'algoritmo di ordinamento per selezione

- Il numero delle visite è dell'ordine di n^2
- Usare la *notazione O-grande* (*big-Oh*, in inglese). Il numero delle visite è $O(n^2)$
- Il numero dei confronti quadruplica quando le dimensioni dell'array raddoppiano
- L'espressione $f(n) = O(g(n))$ significa che f non cresce più rapidamente di g
- Per trasformare un'espressione esatta nella corrispondente notazione O-grande, basta individuare il termine che aumenta più rapidamente, n^2, e ignorare il suo coefficiente costante

Ordinamento per inserimento

- Si suppone che la sequenza iniziale `a[0] a[1] ... a[k]` di un array sia già ordinata

- Aggiungiamo ora a tale sequenza l'elemento `a[1];` tale elemento deve essere inserito prima dell'elemento di valore 11

- Aggiungiamo l'elemento `a[2]`

Ordinamento per inserimento

- Ripetiamo il procedimento, inserendo l'elemento `a[3]`

- Infine, l'elemento `a[4]`. L'ordinamento è completo

File InsertionSorter.java

```
01: /**
02:    La classe seguente realizza l'algoritmo di ordinamento
   per inserimento.
03:
04: */
05: public class InsertionSorter
06: {
07:    /**
08:       Costruisce un ordinatore per inserimento.
09:       @param anArray l'array da ordinare
10:    */
11:    public InsertionSorter(int[] anArray)
12:    {
13:       a = anArray;
14:    }
15:
16:    /**
17:       Ordina l'array gestito da questo ordinatore.
18:    */
```

Continua…

File InsertionSorter.java

```
19:     public void sort()
20:     {
21:        for (int i = 1; i < a.length; i++)
22:        {
23:           int next = a[i];
24:           // Cerca la posizione in cui inserire, spostando in
                    //posizioni di indice superiore tutti gli elementi
   di          //valore maggiore
25:           int j = i;
26:           while (j > 0 && a[j - 1] > next)
27:           {
28:              a[j] = a[j - 1];
29:              j--;
30:           }
31:           // inserisci l'elemento
32:           a[j] = next;
33:        }
34:     }
35:
36:     private int[] a;
37: }
```

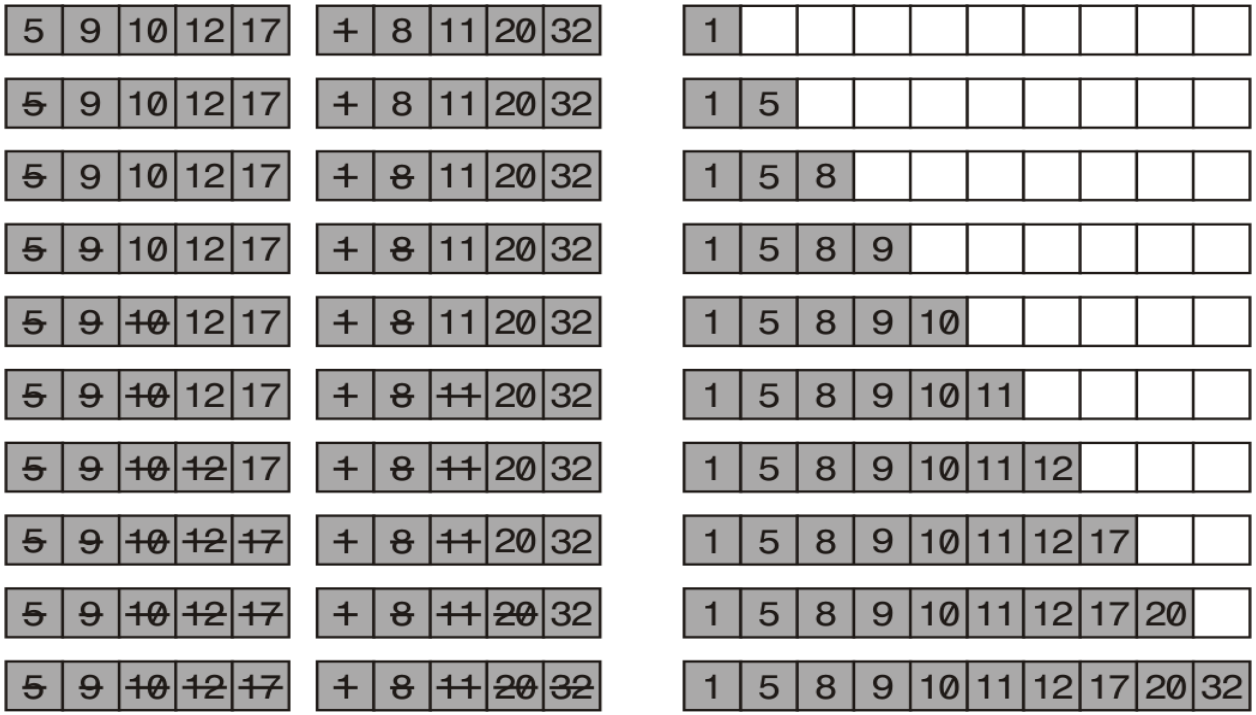
Ordinamento per fusione (MergeSort)

- L’algoritmo di ordinamento per fusione ordina un array:
  - dividendolo a metà
  - ordinando ricorsivamente ciascuna metà
  - fondendo le due metà ordinate
- Questo è più veloce dell’ordinamento per inserimento

Esempio di MergeSort

- Dividere l'array in due metà

- Fondere i due array ordinati in un solo array ordinato



MergeSort

```
public void sort()
   {
      if (a.length <= 1) return;
      int[] first = new int[a.length / 2];
      int[] second = new int[a.length - first.length];
      System.arraycopy(a, 0, first, 0, first.length);
      System.arraycopy(a, first.length, second, 0,
   second.length);
      MergeSorter firstSorter = new MergeSorter(first);
      MergeSorter secondSorter = new
   MergeSorter(second);
      firstSorter.sort();
      secondSorter.sort();
      merge(first, second);
}
```

File MergeSorter.java

```
01: /**
02:    Questa classe ordina un array, usando l'algoritmo
di ordinamento per fusione.
03: */
04: public class MergeSorter
05: {
06:    /**
07:       Costruisce un ordinatore per fusione.
08: @param anArray l'array da ordinare
09:    */
10:    public MergeSorter(int[] anArray)
11:    {
12:       a = anArray;
13:    }
14:
15:    /**
16:   Ordina l'array gestito da questo ordinatore per fusione.
17:    */
```

Continua…

File MergeSorter.java

```
18:    public void sort()
19:    {
20:       if (a.length <= 1) return;
21:       int[] first = new int[a.length / 2];
22:       int[] second = new int[a.length - first.length];
23:       System.arraycopy(a, 0, first, 0, first.length);
24:       System.arraycopy(a, first.length, second, 0,
             second.length);
25:       MergeSorter firstSorter = new MergeSorter(first);
26:       MergeSorter secondSorter = new MergeSorter(second);
27:       firstSorter.sort();
28:       secondSorter.sort();
29:       merge(first, second);
30:    }
31:
```

Continua…

File MergeSorter.java

```
32:     /**
33:        Fonde due array ordinati per generare l'array che
34:     deve essere ordinato da questo ordinatore per fusione.
35:     @param first il primo array ordinato
36:     @param second il secondo array ordinato
37:     */
38:     private void merge(int[] first, int[] second)
39:     {
40:
41: // il prossimo elemento da considerare nel primo array
42:        int iFirst = 0;
43: // il prossimo elemento da considerare nel secondo array
44:         int iSecond = 0;
45: // la prossima posizione libera nell'array a
46:        int j = 0;
47:
48:
```

Continua…

File MergeSorter.java

```
49:         // finché né iFirst né iSecond oltrepassano la fine,
50:         // sposta in a l'elemento minore
51:         while (iFirst < first.length && iSecond <
   second.length)
52:         {
53:            if (first[iFirst] < second[iSecond])
54:            {
55:               a[j] = first[iFirst];
56:               iFirst++;
57:            }
58:            else
59:            {
60:               a[j] = second[iSecond];
61:               iSecond++;
62:            }
63:            j++;
64:         }
65:
```

Continua…

File MergeSorter.java

```
66:          // notate che soltanto una delle due copiature
67:          // seguenti viene eseguita
68:
69:          // copia tutti i valori rimasti nel primo array
70:          System.arraycopy(first, iFirst, a, j,
                first.length - iFirst);
71:
72:          // copia tutti i valori rimasti nel secondo array
73:          System.arraycopy(second, iSecond, a, j,
                second.length - iSecond);
74:       }
75:
76:       private int[] a;
77: }
```

File MergeSortDemo.java

```
01: import java.util.Arrays
   /**
02:    Questo programma collauda l'algoritmo di ordinamento
03:per fusione ordinando un array che contiene numeri casuali.
04: */
05: public class MergeSortDemo
06: {
07:    public static void main(String[] args)
08:    {
09:       int[] a = ArrayUtil.randomIntArray(20, 100);
10:       System.out.println(Arrays.toString(a));
11:       MergeSorter sorter = new MergeSorter(a);
12:       sorter.sort();
13:        System.out.println(Arrays.toString(a));
14:    }
15: }
16:
```

File `MergeSortDemo.java`

- Visualizza

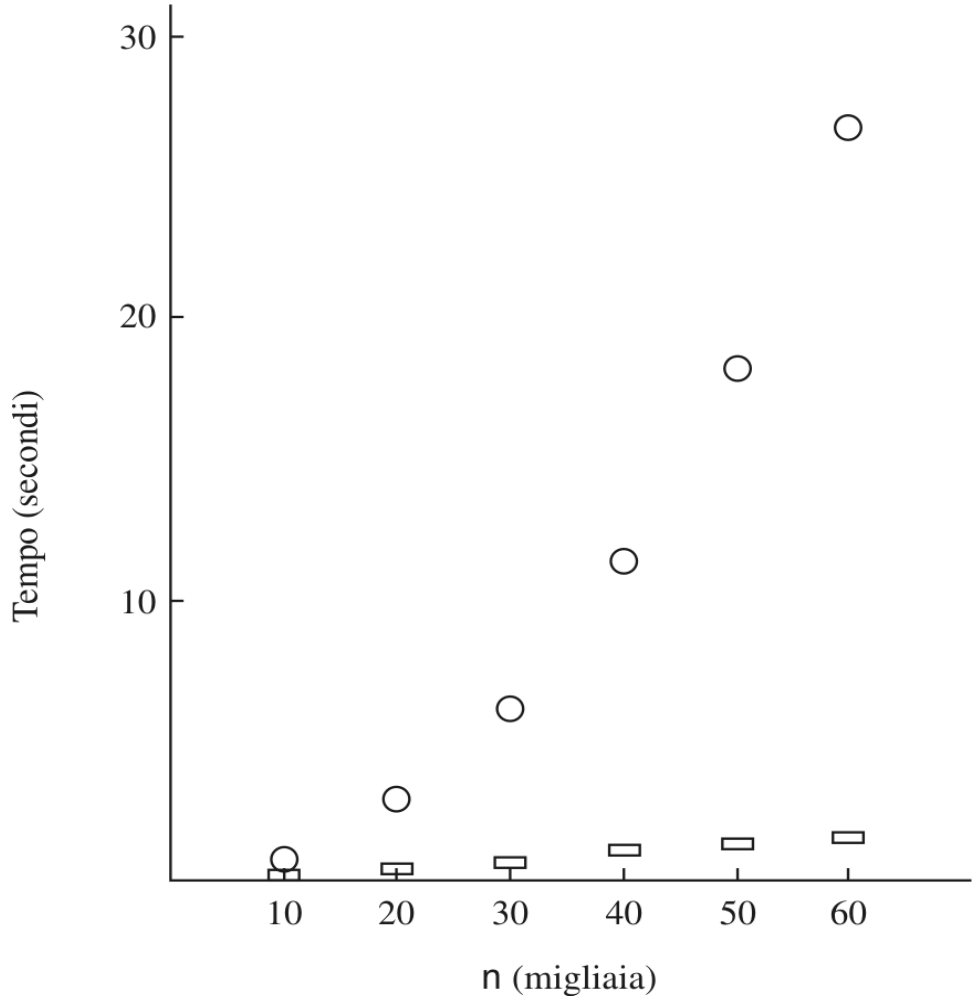
```
 8 81 48 53 46 70 98 42 27 76 33 24 2 76 62 89 90 5 13 21
2 5 8 13 21 24 27 33 42 46 48 53 62 70 76 76 81 89 90 98
```

Analisi dell'algoritmo di ordinamento per fusione

n	Ordinamento per fusione (millisecondi)	Ordinamento per selezione (millisecondi)
10 000	31	772
20 000	47	3051
30 000	62	6846
40 000	80	12 188
50 000	97	19 015
60 000	113	27 359

Tempo di esecuzione dell'ordinamento per fusione vs tempo di esecuzione dell'ordinamento per selezione

Figura 2



Analisi dell'algoritmo di ordinamento per fusione

- Proviamo a stimare il numero di visite agli elementi dell'array
- Supponiamo che n sia una potenza di 2, diciamo $n = 2^m$
- Affrontiamo come prima cosa il processo di fusione che si verifica dopo che la prima e la seconda metà sono state ordinate

  - Conteggiamo questa operazione come 3 visite per ogni elemento, ovvero $3n$ visite
  - Altre $2n$ visite, per creare due sottoarray
  - Totale: $5n$

Analisi dell'algoritmo di ordinamento per fusione

- Se chiamiamo $T(n)$ il numero di visite necessarie per ordinare un array di n elementi
  - $T(n) = T(n/2) + T(n/2) + 5n$ o
  - $T(n) = 2T(n/2) + 5n$
- Valutiamo $T(n/2)$ usando la stessa formula:

  $T(n/2) = 2T(n/4) + 5n/2$. Quindi:
  - $T(n) = 2 \times 2T(n/4) + 5n + 5n$
- Facciamolo di nuovo:

  $T(n/4) = 2T(n/8) + 5n/4$. Quindi:
  - $T(n) = 2 \times 2 \times 2T(n/8) + 5n + 5n + 5n$

Analisi dell'algoritmo di ordinamento per fusione

- Generalizzando da 2, 4, 8 a potenze arbitrarie di 2:

  $T(n) = 2^k T(n/2^k) + 5nk$

- Ricordiamo che abbiamo assunto $n = 2^m$; di conseguenza, per $k = m$,

  $T(n) = 2^m T(n/2^m) + 5nm$

  $= nT(1) + 5nm$

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

Analisi dell'algoritmo di ordinamento per fusione

- Per determinare l'ordine di crescita della funzione

  - eliminiamo il termine di grado inferiore, n

  - eliminiamo il fattore costante 5

  - eliminiamo anche la base del logaritmo, perché tutti i logaritmi sono correlati da un fattore costante

  - rimaniamo con l'algoritmo $O(n \log(n))$.

- Usando la notazione O-grande il numero delle visite è:

  $O(n \log(n))$.

Ordinamento per fusione vs. ordinamento per selezione

- L'ordinamento per selezione è un algoritmo O(n^2)
- L'ordinamento per fusione è un algoritmo O(n log(n)).
- La funzione n log(n) cresce molto più lentamente di n^2.

Scrivere programmi in Java

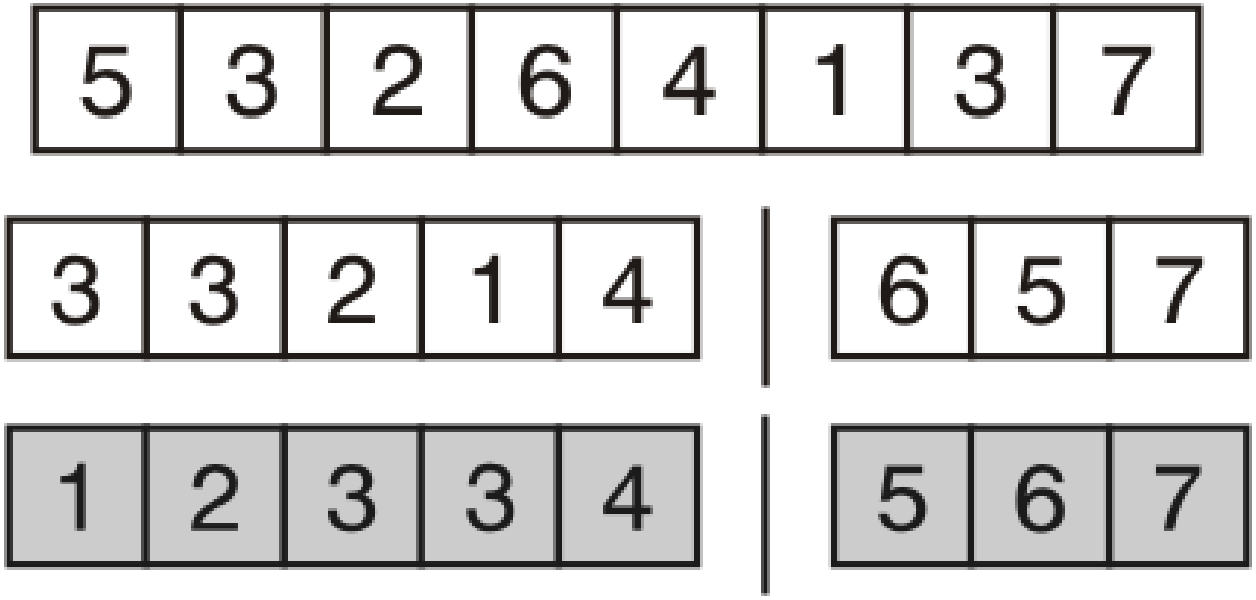
- La classe `Arrays` realizza il metodo di ordinamento che dovreste sempre usare nei vostri programmi Java
- Potete ordinare un array di numeri interi scrivendo semplicemente così:

```
int[] a = ...;
Arrays.sort(a);
```

- Tale metodo `sort` usa l'algoritmo `Quicksort` (vedi Argomenti avanzati 13.3)

L'algoritmo `Quicksort`

- Dividere per vincere
  - Suddivisione o partizionamento della porzione
  - Ordinamento della porzione



L'algoritmo Quicksort

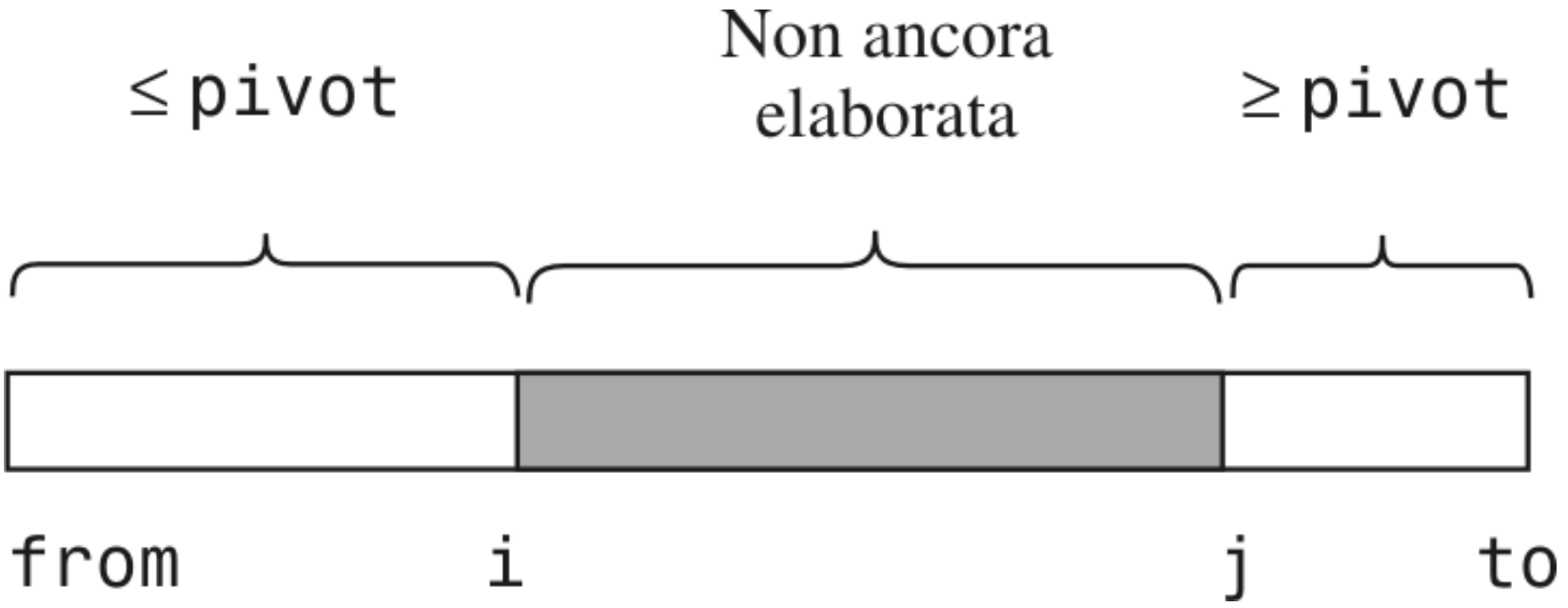
```
public void sort(int from, int to)

{
      if (from >= to) return;
      int p = partition(from, to);
      sort(from, p);
      sort(p + 1, to);

}
```

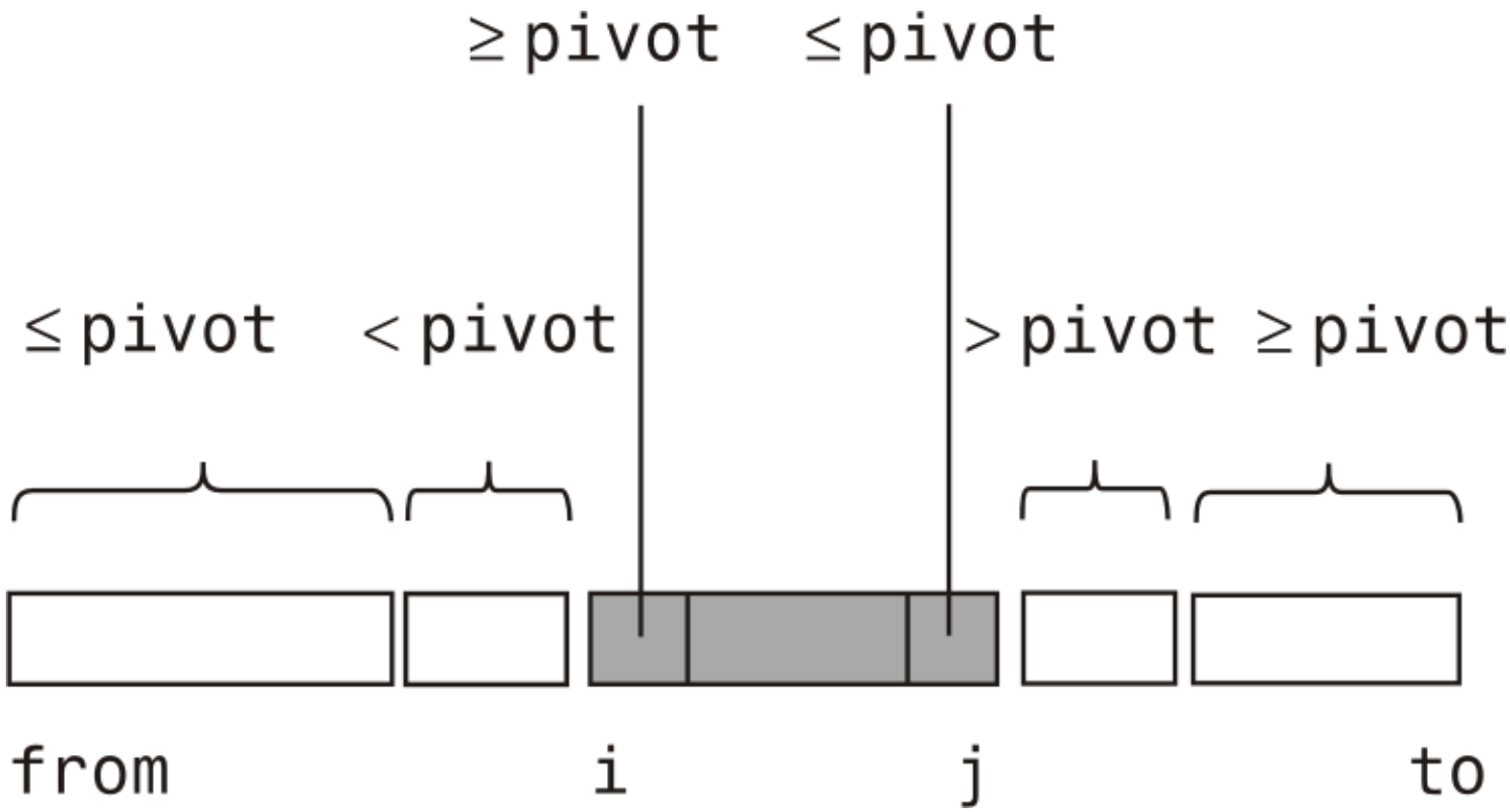
L'algoritmo `Quicksort`

- Suddividere una porzione



L'algoritmo `Quicksort`

- Estendere le porzioni

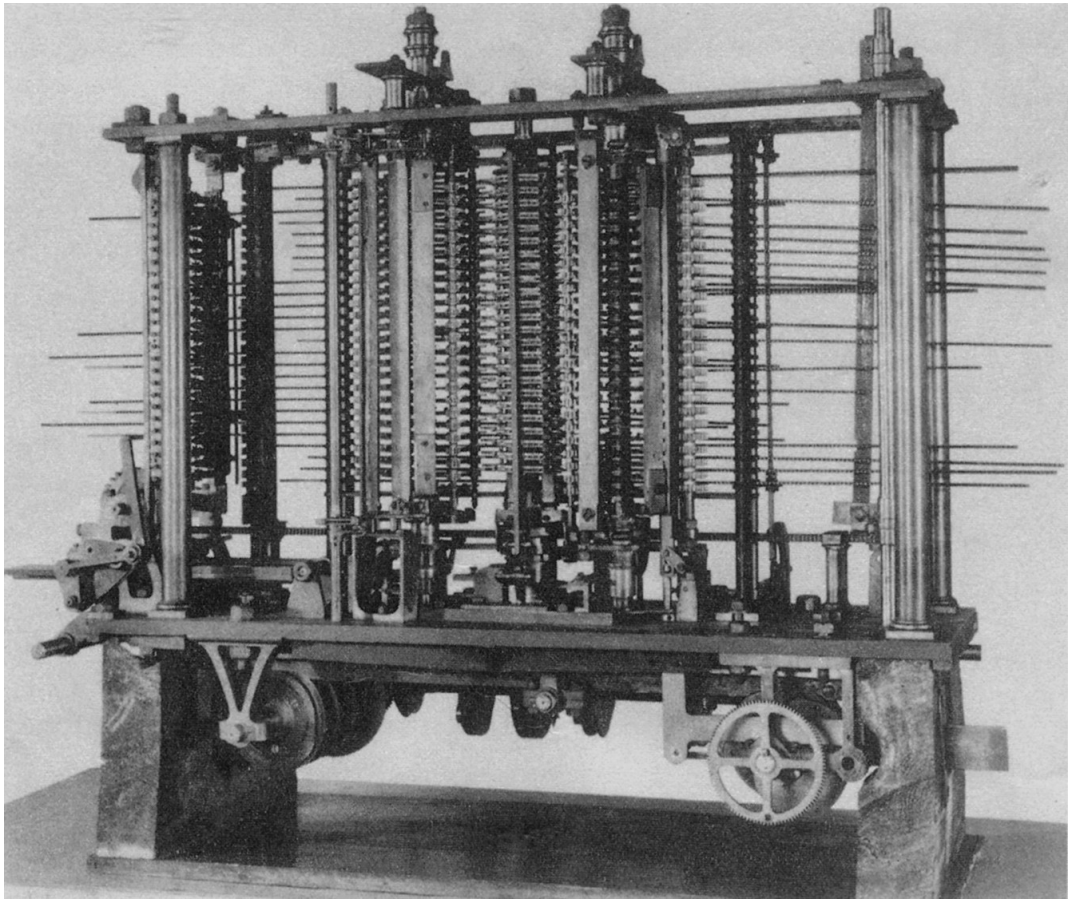


L'algoritmo Quicksort

```
private int partition(int from, int to)
   {
      int pivot = a[from];
      int i = from - 1;
      int j = to + 1;
      while (i < j)
      {
         i++; while (a[i] < pivot) i++;
         j--; while (a[j] > pivot) j--;
         if (i < j) swap(i, j);
         }
      return j;
   }
```

Il primo programmatore

Figura 5:
Difference Engine
di Babbage



Effettuare ricerche

- Ricerca lineare o sequenziale
- Una ricerca lineare esamina tutti i valori di un array finché trova una corrispondenza con quanto cercato, oppure raggiunge la fine dell'array
- Nell'ipotesi che l'elemento v sia presente nell'array a di lunghezza *n*, la ricerca richiede in media la visita di *n*/2 elementi
- Una ricerca lineare trova un valore in un array in O(n) passi.

File LinearSearcher.java

```
01: /**
02:    Una classe per eseguire ricerche lineari in un array.
03: */
04: public class LinearSearcher
05: {
06:    /**
07:       Costruisce l'oggetto di tipo LinearSearcher.
08:       @param anArray un array di numeri interi
09:    */
10:    public LinearSearcher(int[] anArray)
11:    {
12:       a = anArray;
13:    }
14:
15:    /**
16:       Trova un valore in un array usando l'algoritmo
17:       di ricerca lineare.
```

Continua…

File LinearSearcher.java

```
18:          @param v il valore da cercare
19:          @return l'indice in cui si trova il valore,oppure -1
20:          se non è presente nell'array
21:       */
22:       public int search(int v)
23:       {
24:          for (int i = 0; i < a.length; i++)
25:          {
26:             if (a[i] == v)
27:                return i;
28:          }
29:          return -1;
30:       }
31:
32:       private int[] a;
33: }
```

File LinearSearchDemo.java

```
01: import java.util.Scanner;
02:
03: /**
04:  Questo programma utilizza l'algoritmo di ricerca lineare.
05: */
06: public class LinearSearchDemo
07: {
08:    public static void main(String[] args)
09:    {
10:
11:
12:       int[] a = ArrayUtil.randomIntArray(20, 100);
13:       ArrayUtil.print(a);
14:       LinearSearcher searcher = new LinearSearcher(a);
15:
16:       Scanner in = new Scanner(System.in);
17:
```

Continua…

File LinearSearchDemo.java

```
18:         boolean done = false;
19:         while (!done)
20:         {
21:            System.out.print("Enter number to search for,
                  -1 to quit: ");
22:            int n = in.nextInt();
23:            if (n == -1)
24:               done = true;
25:            else
26:            {
27:               int pos = searcher.search(n);
28:               System.out.println("Found in position " + pos);
29:            }
30:         }
31:      }
32:   }
```

File `LinearSearchDemo.java`

- Visualizza

```
46 99 45 57 64 95 81 69 11 97 6 85 61 88 29 65 83 88 45 88
Enter number to search for, -1 to quit: 11
Found in position 8
```

Ricerca binaria

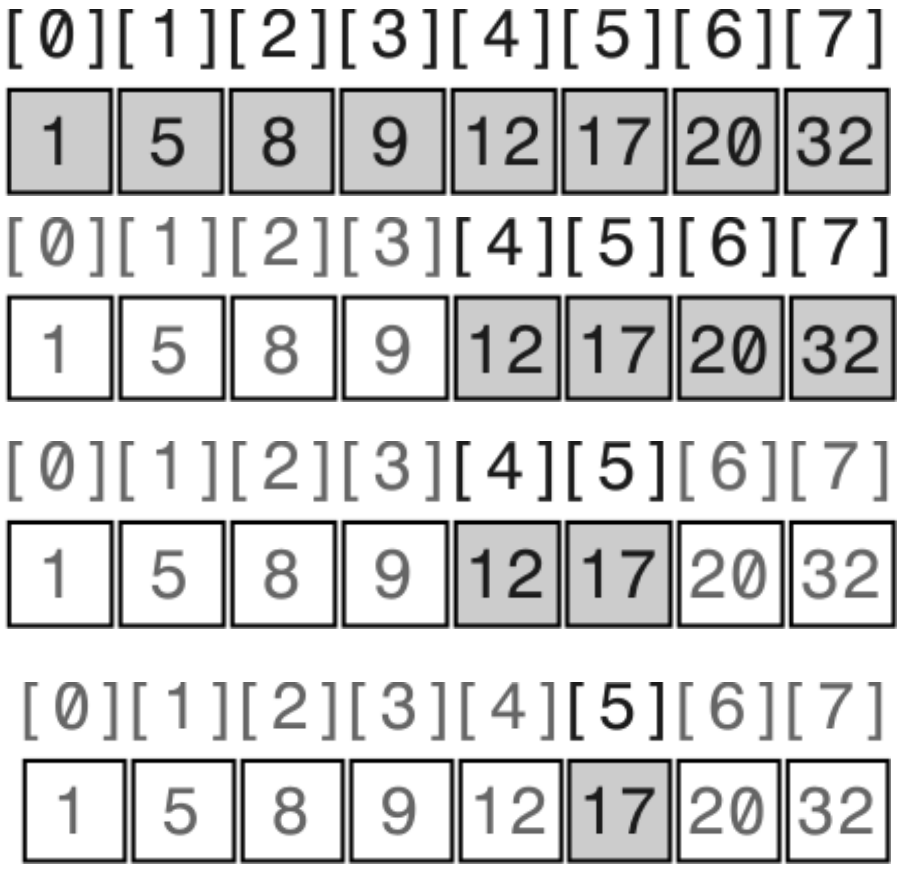
- Una ricerca binaria cerca un valore in un array ordinato

  - Determinando se il valore si trova nella prima o nella seconda metà dell'array

  - Ripetendo poi la ricerca in una delle due metà.

Ricerca binaria

non abbiamo trovato il numero cercato perché 15 è diverso da 17.



File BinarySearcher.java

```
01: /**
02:    Una classe per eseguire ricerche binarie in un array.
03: */
04: public class BinarySearcher
05: {
06:    /**
07:       Costruisce un oggetto di tipo BinarySearcher.
08:       @param anArray un array ordinato di numeri interi
09:    */
10:    public BinarySearcher(int[] anArray)
11:    {
12:       a = anArray;
13:    }
14:
15:    /**
16:       Trova un valore in un array ordinato,
17:       utilizzando l'algoritmo di ricerca binaria.
```

Continua…

File `BinarySearcher.java`

```
18:         @param v il valore da cercare
19:         @return l'indice della posizione in cui si trova
20:         il valore, oppure -1 se non è presente
21:     */
22:     public int search(int v)
23:     {
24:         int low = 0;
25:         int high = a.length - 1;
26:         while (low <= high)
27:         {
28:             int mid = (low + high) / 2;
29:             int diff = a[mid] - v;
30:
31:             if (diff == 0) // a[mid] == v
32:                 return mid;
33:             else if (diff < 0) // a[mid] < v
34:                 low = mid + 1;
```

Continua…

File BinarySearcher.java

```
35:             else
36:                high = mid - 1;
37:          }
38:          return -1;
39:       }
40:
41:       private int[] a;
42: }
43:
```

Ricerca binaria

- Proviamo a stabilire quante visite di elementi dell'array sono necessarie per portare a termine una ricerca
- esaminiamo l'elemento di mezzo e poi esploriamo o il sottoarray di sinistra o quello di destra:

  $T(n) = T(n/2) + 1$
- Utilizzando la stessa equazione, si ha:

  $T(n/2) = T(n/4) + 1$
- Inserendo questo risultato nell'equazione originale, otteniamo:

  $T(n) = T(n/4) + 2$
- Generalizzando, si ottiene:

  $T(n) = T(n/2^k) + k$

Ricerca binaria

- Facciamo l'ipotesi semplificativa che n sia una potenza di 2, $n = 2m$, dove $m = \log_2(n)$.
- Otteniamo quindi

  $T(n) = 1 + \log_2(n)$

  Di conseguenza, la ricerca binaria è un algoritmo $O(\log(n))$.

Ricerca di un array

- La classe `Arrays` contiene un metodo statico `binarySearch`
- Se un valore non viene trovato nell'array, il valore restituito non è –1, ma –*k*–1, dove *k* è la posizione prima della quale andrebbe inserito l'elemento

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
   // restituisce -3
   // v andrebbe inserito prima della posizione 2
```

Ordinare dati veri

- Il metodo `sort` della classe `Arrays` ordina oggetti di classi che realizzano l'interfaccia `Comparable`

```
public interface Comparable
{
   int compareTo(Object otherObject);
}
```

- L'invocazione `a.compareTo(b)` deve restituire:
  - un numero negativo se `a` precede `b`
  - 0 se `a` e `b` sono uguali
  - un numero positivo se `a` segue `b`

Ordinare dati veri

- Molte classi della libreria standard di Java, come `String` e `Date`, realizzano l'interfaccia `Comparable`
- Potete realizzare l'interfaccia `Comparable` anche nelle vostre classi

```
public class Coin implements Comparable
{
   . . .
   public int compareTo(Object otherObject)
   {
      Coin other = (Coin) otherObject;
      if (value < other.value) return -1;
      if (value == other.value) return 0;
      return 1;
   }
   . . .
}
```

Il metodo `CompareTo`

- Il metodo definisce una *relazione d'ordine totale*, con le tre seguenti proprietà:
  - *Antisimmetrica*: Se `a.compareTo(b) ≤ 0`, allora `b.compareTo(a) ≥ 0`
  - *Riflessiva*: `a.compareTo(a) = 0`
  - *Transitiva*: Se `a.compareTo(b) ≤ 0` e `b.compareTo(c) ≤ 0`, allora `a.compareTo(c) ≤ 0`

Ordinare dati veri

- Una volta che la vostra classe `Coin` realizza l'interfaccia `Comparable`, potete semplicemente usare un array di monete come argomento del metodo `Arrays.sort`

```
Coin[] coins = new Coin[n];
// Add coins
. . .
Arrays.sort(coins);
```

Ordinare dati veri

- Se le monete sono memorizzate in un oggetto di tipo `ArrayList`, utilizzate invece il metodo `Collections.sort`, che usa l'algoritmo di ordinamento per fusione:

```
Collections.sort: ArrayList<Coin> coins = new ArrayList<Coin>();
// aggiungi monete
. . .
Collections.sort(coins);
```