DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Gestione di file in Java

Gestione di file in Java

- Finora abbiamo visto programmi Java che interagiscono con l'utente soltanto tramite i flussi di ingresso e di uscita
- Ora vediamo
  - Come ciascuno tali flussi può essere collegato a un file con un comando di sistema operativo (redirezione di input e di output)
  - Come si può leggere e scrivere file **all'interno** di un programma Java, in modo esplicito

Reindirizzamento di input e output

Calcolare la somma di numeri

```
import java.util.Scanner;

public class Sum
{
    public static void main(String[] args)
   {
       Scanner console = new Scanner(System.in);
       double sum = 0;
       while (console.hasNextDouble()){
          sum = sum + console.nextDouble();
       }
       System.out.println("Somma: " + sum);
       console.close();
    }
}
```

Reindirizzamento di input e output

- Usando il programma **Sum** (Sum.java) si inseriscono dei numeri da tastiera, che al termine non vengono memorizzati
  - ***per sommare una serie di numeri, bisogna digitarli tutti, ma non ne rimane traccia! Se si fa un errore…***
- Alternativa interessante e utile:
  il programma legge i numeri da un file
  - questo si può fare con il **reindirizzamento dell'input standard**

Reindirizzamento di input e output

- Il reindirizzamento dell'input standard, sia nei sistemi Unix sia nei sistemi Microsoft Windows, si indica con il carattere **<** seguito dal **nome del file da cui ricevere l'input**

```
java Sum < numeri.txt
```

- Si dice che il file **numeri.txt** viene ***collegato*** all'input standard
- Il programma non ha bisogno di alcuna istruzione particolare, semplicemente **System.in** non sarà più collegato alla tastiera ma al file specificato
  - La tastiera "non funziona più"… non si possono introdurre dati *anche* dalla tastiera

Reindirizzamento di input e output

- A volte è comodo anche il reindirizzamento dell'output
  - ad esempio, quando il programma produce molte righe di output, che altrimenti scorrono velocemente sullo schermo senza poter essere lette

```
java Sum > output.txt
```

    - Se il file output.txt non esiste viene creato, se esiste viene sovrascritto
    - Per non sovrascrivere il file di output bisogna utilizzare:

```
java Sum >> output.txt
```

- I due reindirizzamenti possono anche essere combinati

```
java Sum < numeri.txt > output.txt
```

Canalizzazioni ("pipes")

Canalizzazioni (pipes) un esempio

- Supponiamo di aver scritto la classe Split che scrive ciascuna parola ricevuta in ingresso su una riga di output separata

```
java Split < testo.txt
```

- Una elaborazione molto comune consiste nell'ordinare poi tali parole questa elaborazione è talmente comune che quasi tutti i sistemi operativi hanno un programma sort
- in alternativa, possiamo scrivere una classe Sort

Canalizzazioni (pipes)

- Per ottenere le parole di **testo.txt** una per riga e ordinate, abbiamo bisogno di un ***file temporaneo***

```
java Split < testo.txt > temp.txt
sort < temp.txt > testoOrdinato.txt
```

- Il file temporaneo **temp.txt** ***serve soltanto per memorizzare il risultato intermedio***, prodotto dal primo programma e utilizzato dal secondo
- Questa situazione è talmente comune che quasi tutti i sistemi operativi offrono un'alternativa

Canalizzazioni (pipes)

- Anziché utilizzare un file temporaneo per memorizzare l'output prodotto da un programma che deve servire da input per un altro programma, si usa una canalizzazione ("pipe")

```
java Split < testo.txt | sort > testoOrdinato.txt
```

- La canalizzazione può anche prolungarsi… ad esempio, possiamo eliminare eventuali parole ripetute da testoOrdinato.txt

```
java Split < testo.txt | sort | java Unique > out.txt
```

```
java Sort
```

oppure

Formattazione di numeri

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Formattazione di numeri

- Non sempre il formato standard per stampare numeri corrisponde ai nostri desideri

```
double total = 3.50;
final double TAX_RATE = 8.5; // aliquota
d'imposta in percentuale
double tax = total * TAX_RATE / 100;
System.out.println("Total: " + total);
System.out.println("Tax: " + tax);
```

- Ci piacerebbe di piu' visualizzare i numeri
  - Con due cifre decimali
  - Incolonnati

```
Total: 3.5
Tax: 0.2975
```

```
Total:   3.50
Tax:     0.30
```

Formattazione di numeri

- Java fornisce il metodo **printf**
  - Il primo parametro esplicito di **printf** è una **stringa di formato** che contiene dei caratteri da stampare e degli **specificatori di formato**
    - Ogni specificatore di formato comincia con il carattere %
  - I parametri successivi sono i **valori da visualizzare** secondo i formati specificati

```
System.out.printf("Total:%5.2f", total);
```

- Produce:

```
Total: 3.50
```

spazio

Formattazione di numeri

- **%5.2f** è lo specificatore di formato: numero **in virgola mobile** (**%f**) formato da **5 caratteri** (compreso il punto!) con **due cifre dopo la virgola**
- Questo formato viene applicato alla variabile total, che è il secondo parametro del metodo

Tipi di formato e modificatori di formato

Codice	Tipo	Esempio
d	Intero decimale	123
x	Intero esadecimale	7B
o	Intero ottale	173
f	Virgola mobile	12.30
e	Virgola mobile esponenziale	1.23e+1
g	Virgola mobile generico (notazione esponenziale per i numeri molto grandi o molto piccoli)	12.3
s	Stringa	Tax:
n	Fine riga indipendente dalla piattaforma	

Codice	Significato	Esempio
-	Allinea a sinistra	1.23 seguito da spazi
0	Mostra gli zeri iniziali	001..23
+	Mostra il segno più per numeri positivi	+1.23
(	Racchiude tra parentesi i numeri negativi	(1.23)
,	Mostra il separatore di migliaia	12,300
^	Usa lettere maiuscole	12.3E+1

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Gestione di file all'interno di un programma

Gestione di file in Java

- Limitiamoci ad affrontare il problema della ***gestione di file di testo*** (file contenenti caratteri, ovviamente codificati in binario)
  - esistono anche i ***file binari***, che contengono semplicemente configurazioni di bit che rappresentano qualsiasi tipo di dati
    - argomento complesso, non ce ne occupiamo
- La gestione dei file avviene interagendo con il sistema operativo mediante classi del pacchetto **java.io** della libreria standard

Leggere un file di testo

Apertura di file di testo

- Prima di leggere caratteri da un file (esistente) occorre ***aprire*** il file in lettura
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileReader**

```
FileReader reader = new FileReader("file.txt");
```

  - il costruttore necessita del nome del file (con eventuale percorso) sotto forma di stringa
  - se il file non esiste, viene lanciata l'eccezione **FileNotFoundException**, che **deve essere obbligatoriamente gestita**

Lettura da file con Scanner

- Si costruisce un'esemplare di **Scanner** che scansiona il file esattamente come farebbe con il flusso di ingresso standard o con una stringa

```
try {
   FileReader reader = new FileReader("file.txt");
   Scanner sc = new Scanner(reader);
   while(sc.hasNextLine()){
      String s = sc.nextLine(); // ad esempio
      ...  // elabora s
   }
}
catch (FileNotFoundException e){
      System.out.println(...);
}
```

Chiusura dello Scanner

- Al termine della lettura del file (che non necessariamente deve procedere fino alla fine…) occorre *chiudere* il file

```
FileReader reader = new FileReader("file.txt");
...
try { reader.close(); }
catch (IOException e) {...}
```

- Questo metodo può lanciare **IOException**, da gestire obbligatoriamente
- Se il file non viene chiuso non si ha un errore, ma una potenziale situazione di instabilità per il sistema operativo e uno spreco di risorse (memoria)

Eccezioni

- FileNotFoundException è una sottoclasse di **IOException**
- Con IOException si gestisce implicitamente anche FileNotFoundException
- Se voglio distinguerli devo prima gestire **FileNotFoundException**

Scrittura di file di testo (1/3)

- Per scrivere dati in un file si costruisce un oggetto di tipo **PrintWriter**
- Il costruttore necessita del nome del file sotto forma di stringa
- L'oggetto di tipo **PrintWriter** ha i metodi **print/println** che permettono di scrivere comodamente sul file

```
PrintWriter out = new PrintWriter("file.txt");
out.println("Ciao");
```

Scrittura di file di testo (2/3)

- ***Se il file non esiste, viene creato***

- ***Se il file esiste, il suo contenuto viene SOVRASCRITTO con i nuovi contenuti, eventuali contenuti preesistenti vanno perduti!***

Scrittura di file di testo

- Al termine della scrittura del file occorre *chiudere* il file

```
PrintWriter out = new PrintWriter("file.txt");
...
out.close();
```

- Anche questo metodo può lanciare **Exception**, da gestire obbligatoriamente
- ***Se `close` non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto***

try-with-resources

```
try(PrintWriter out = new PrintWriter(filename)){
    …
    out.println("prova");
    …

} // qui viene invocato out.close() anche se e' stata lanciata
  // un'eccezione

catch(…Exception e){

    // codice eseguito se e' stata lanciata l'eccezione del
    // tipo dichiarato nell'argomento del catch
}
finally{
    // codice eseguito in ogni caso
}
```

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Take home message
Gestione di file in Java

- Usando le classi **FileReader** e **PrintWriter** del pacchetto **java.io** è quindi possibile manipolare, all'interno di un programma Java, più file in lettura e/o più file in scrittura
  - Ci sono molte altre possibilità di manipolazione di file rispetto a quanto visto qui
- Rimane invariata la possibilità di utilizzare i flussi di ingresso e di uscita standard, che possono in realtà essere collegati a file senza che il programma Java ne sia consapevole (redirezione)

Ricorsione

Obiettivi

- Conoscere i principali aspetti della ricorsione
- Imparare a progettare funzioni ricorsive
- Esempi di:
  - Ricorsione numerica
  - Ricorsione doppia
  - Ricorsione strutturale

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…

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

0! = 1

1! = 1(1-1)! = 1·0! = 1·1 = 1

2! = 2(2-1)! = 2·1! = 2·1 = 2

3! = 3(3-1)! = 3·2! = 3·2·1 = 6

4! = 4(4-1)! = 4·3! = 4·3·2·1 = 24

5! = 5(5-1)! = 5·4! = 5·4·3·2·1 = 120

- Quindi, per ogni **n** intero ***positivo***, il fattoriale di **n** è il prodotto dei primi **n** numeri interi positivi
  - Questa proprietà, che discende dalla definizione, viene a volte usata come definizione alternativa

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 // è OK per n = 1 ? Casi limite...
   {  int p = 1;
      for (int i = 2; i <= n; i++){
         p = p * i;
      }
      return p;
   } // questi else in realtà non servono, perché?
}
```

Clausola else a volte inutile

```
if (...)
   return ...; // oppure throw ...
else
   ...
```

- Ci sono casi in cui una clausola **else**, presente nel flusso logico del programma, è in realtà inutile dal punto di vista tecnico (e, in pratica, non si mette)
  - La clausola **else** serve a escludere l'esecuzione di un blocco di codice quando il corpo del corrispondente **if** viene eseguito
  - Quando, però, l'esecuzione del corpo dell'**if** provoca la terminazione del metodo (naturale, con **return**, o prematura, con **throw**), gli enunciati che seguono l'**if** sono conseguentemente esclusi dalla possibilità di essere eseguiti, quindi la clausola **else** non serve

Il calcolo del fattoriale

- Fin qui, nulla di nuovo… però abbiamo dovuto fare un'analisi matematica della definizione per scrivere l'algoritmo, dopo aver scoperto che, per calcolare *n*!, occorre fare la moltiplicazione dei primi *n* numeri interi
- Realizzando direttamente la definizione

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

sarebbe stato più naturale scrivere

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

Il calcolo del fattoriale

```
public static int factorial(int n)
{   if (n < 0)
       throw new IllegalArgumentException();
    if (n == 0)
       return 1;
    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 sperimentalmente** scrivendo un programma che usi il metodo **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 una tecnica algoritmica che si chiama

  ricorsione

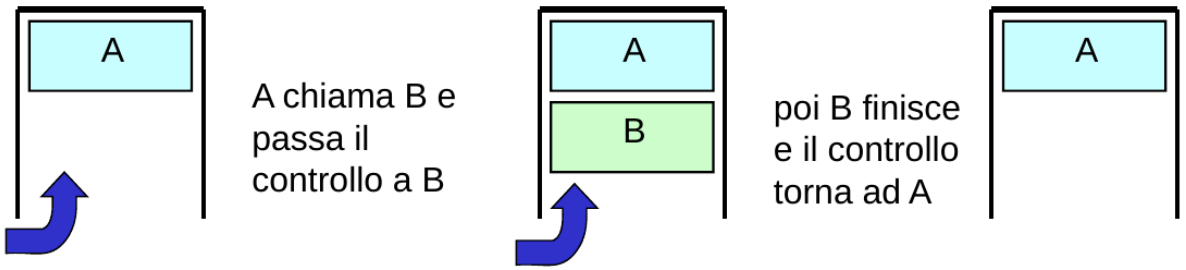
  e un metodo che ne faccia uso si chiama

  metodo ricorsivo

- La ricorsione è uno strumento molto potente per realizzare alcuni algoritmi e produce generalmente codice molto "pulito ed elegante", ma è anche fonte di **errori di difficile diagnosi**: **va usata con cautela**

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



CODICE

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

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

La ricorsione

Vediamo la sequenza usata per calcolare 3!

si invoca factorial(3)

factorial(3) invoca factorial(2)

factorial(2) invoca factorial(1)

factorial(1) invoca factorial(0)

factorial(0) restituisce 1

factorial(1) restituisce 1*1 = 1

factorial(2) restituisce 2*1 = 2

factorial(3) restituisce 3*2 = 6

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Progettazione di metodi ricorsivi

Progettazione di metodi ricorsivi

- **Si scompone il problema** in una porzione "semplice" e in un'altra parte, **simile a quella iniziale** ma di "dimensione" inferiore
- Nel metodo, si eseguono le fasi seguenti (la prima e la seconda fase possono scambiarsi, dipende dal problema)
  - Risolvere la porzione semplice
    - **A volte c'è solo questa fase (es. se `n < 1`)** e allora si dice che la ricorsione si trova in un suo "caso base"
  - Effettuare un'invocazione ricorsiva per risolvere la porzione complessa (es: calcolare **`factorial(n-1)`**)
    - **Importante: non ci si preoccupa di come verrà risolta questa parte, altrimenti non si riesce a comprendere il funzionamento del metodo**
  - Comporre la soluzione usando i risultati intermedi delle fasi precedenti (es: calcolare **`n * factorial(n-1)`**)

La ricorsione: caso base

- **Prima regola (condizione necessaria)**
  - il metodo ricorsivo ***deve fornire la soluzione del problema in almeno un caso particolare, senza ricorrere a un'invocazione ricorsiva***
    - tale caso si chiama ***caso base*** della ricorsione
  - nel nostro esempio, il caso base era

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

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

La ricorsione: passo ricorsivo

- **Seconda regola (condizione necessaria)**
  - il metodo ricorsivo ***deve effettuare l'invocazione 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**

```
// eseguita se n > 0
   return n * factorial(n - 1);
```

  - il concetto di "problema più semplice" varia di volta in volta: in generale, ***bisogna avvicinarsi 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 invocazioni ricorsive si possano succedere una dopo l'altra, all'infinito; invece, se
  - ad 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

- ***se manca il caso base***, il metodo ricorsivo continua a invocare se stesso "all'infinito"
- ***se il problema non viene semplificato ad ogni invocazione ricorsiva***, il metodo ricorsivo continua a 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 il più semplice programma che presenta ricorsione infinita

```
public class InfiniteRecursion
{  public static void main(String[] args)
   {  System.out.print("+"); // ad esempio
      main(args);
   } // violate entrambe le regole!
}
```

- Il programma terminerà con la segnalazione dell'eccezione **StackOverflowError**
  - il ***runtime stack*** è la struttura che gestisce le invocazioni in attesa all'interno dell'interprete Java
  - ***Overflow*** ricordiamo significa ***trabocco***

Eliminare la ricorsione

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

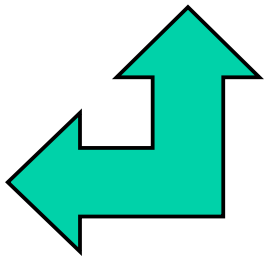
```
public void tail(...)
{   ...
    tail(...);
}
```

Eliminare la ricorsione in coda

- La ricorsione in coda può sempre essere agevolmente ***eliminata***, trasformando il metodo ricorsivo in un metodo che usa un ***ciclo***

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

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



Eliminare 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 (per esempio nel calcolo della funzione fattoriale)
- In ogni caso, la ricorsione in coda è ***meno efficiente*** del ciclo equivalente, perché il sistema deve gestire le invocazioni sospese

Ricorsione multipla

La ricorsione multipla

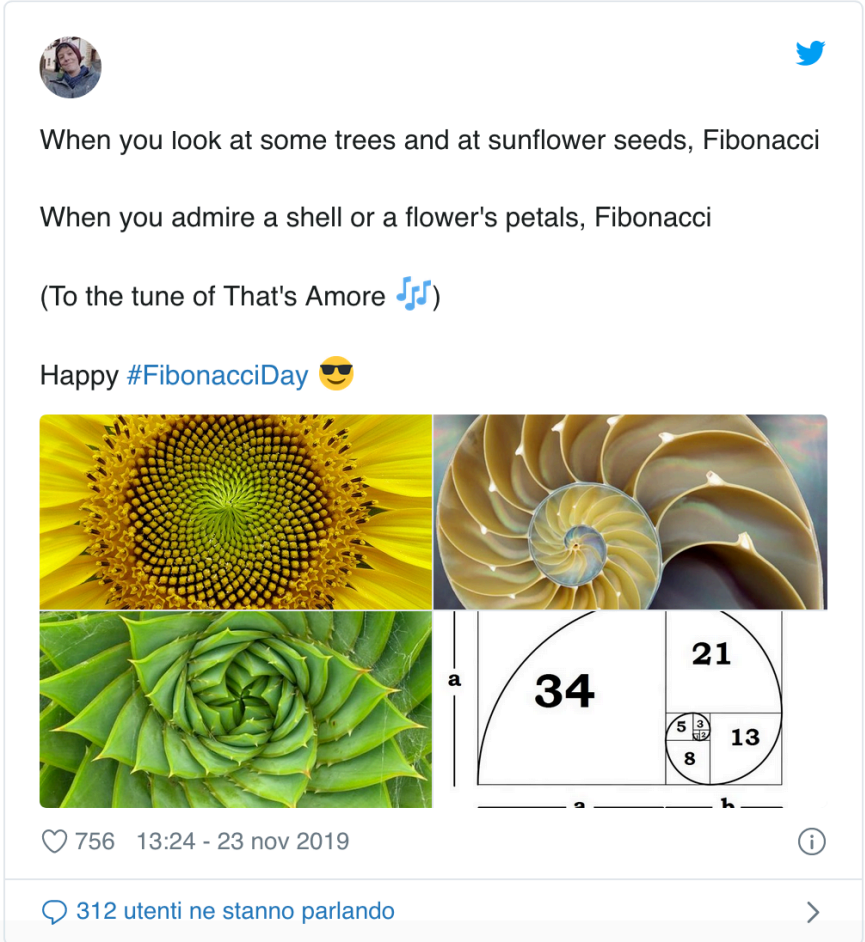
- Si parla di ricorsione multipla (contrapposta alla ricorsione semplice o *lineare*) quando un metodo invoca se stesso **più volte** durante una sua singola esecuzione
- Esempio: il calcolo dei numeri di Fibonacci

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

- 0 1 1 2 3 5 8 13 21 34 …

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Fibonacci's day: 11/23



La ricorsione multipla

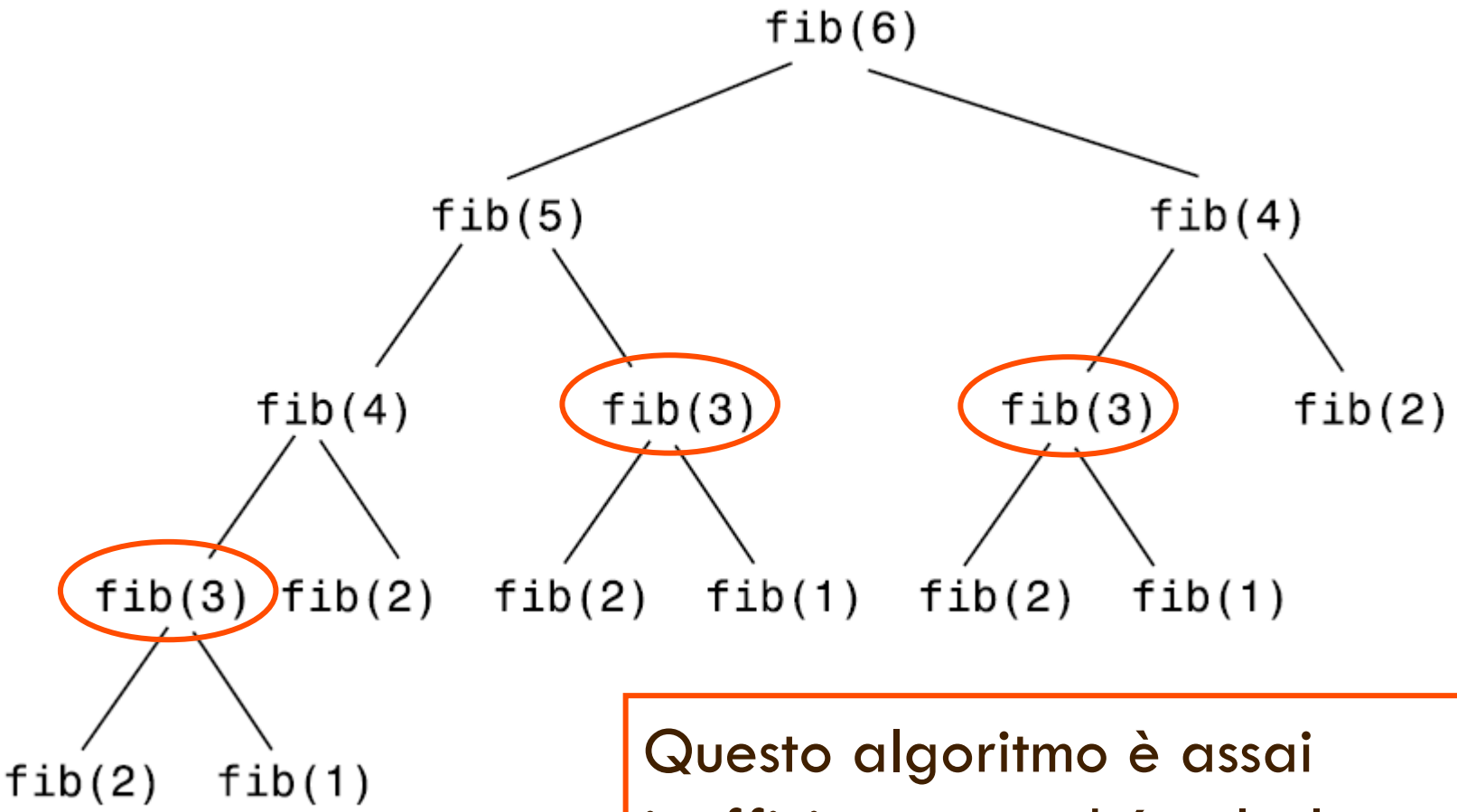
$$\text{Fib}(n) = \begin{cases} n & \text{se } 0 \le n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{se } n \ge 2 \end{cases}$$

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

    if (n < 2)
        return n;

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

} // ricorsione multipla (in particolare, doppia)
```

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… servono quasi 3 milioni di invocazioni per calcolare Fib(31) !!!!
    - **Basterebbe eseguire 30 addizioni!!**
- Attenzione: Fib(47) non sta in un **int**…

La ricorsione multipla



Questo algoritmo è assai inefficiente, perché calcola molte volte gli stessi valori!!

Calcolo iterativo di Fibonacci

- Per calcolare **`Fib(n)`** sono sufficienti **`n-1`** addizioni

```
public static int iterativeFib(int n)
{  if (n < 0) throw new IllegalArgumentException();
   if (n < 2) return n; // in realtà basta n < 1
   int fib0 = 0;
   int fib1 = 1;
   for (int i = 2; i <= n; i++)
   {  int newFib = fib0 + fib1;
      fib0 = fib1;
      fib1 = newFib;
   }
   return fib1;
}
```

- Sul mio computer (Intel iCore7 a 3.1 GHz si calcola Fib(45))
  - In circa **5 secondi** con il metodo ricorsivo
  - In un tempo non misurabile (risultato "istantaneo") con il metodo iterativo

Ricorsione strutturale

Ricorsione funzionale e strutturale

- La ricorsione vista finora si dice anche ***funzionale*** e viene usata per calcolare funzioni (solitamente matematiche)
  - Fattoriale di un numero, Sequenza di Fibonacci, ecc.

- La ricorsione ***strutturale*** sfrutta, invece, caratteristiche intrinseche della struttura di memorizzazione dei dati di un problema
  - Ad esempio, un array di `n` elementi può essere visto come la concatenazione di un elemento e di un array di `n-1` elementi
    - Una diversa suddivisione strutturale spesso utilizzata nella ricorsione operante su array identifica due sotto-problemi relativi a due metà dell'array stesso
  - Il medesimo ragionamento è valido per una stringa

Esercizio: Inversione di una stringa

- Per invertire una stringa usando un algoritmo ricorsivo possiamo immaginare di
  - Prendere l'ultimo carattere della stringa da invertire
  - Generare la stringa inversa come concatenazione di tale carattere e della stringa inversa della parte rimanente

- L'equivalente soluzione iterativa, che abbiamo già visto, è altrettanto semplice

```
public static String reverse(String s){
    if(s == null)
      throw IllegalArgumentException;
    if (s.length() < 2)
      return s; // caso base

    // invocazione con un problema più semplice
    return s.charAt(s.length()-1)
        + reverse(s.substring(0,s.length()-1));

}
```

reverse(CIAO) invoca reverse(CIA)

reverse(CIA) invoca reverse(CI)

reverse(CI) invoca reverse(C)

reverse(C) return C

reverse(CI) return I+C (=IC)

reverse(CIA) return A+ IC (=AIC)

Reverse(CIAO) return O+AIC =(OAIC)

Take home message

- Abbiamo imparato un approccio algoritmico alternativo alle risoluzioni iterative: la ricorsione
- La progettazione di un algortimo ricorsivo richiede l'identificazione di 3 passi
  - Risolvere la porzione semplice
  - Effettuare un'invocazione ricorsiva per risolvere la porzione complessa
  - Comporre la soluzione usando i risultati intermedi delle fasi precedenti
- Esiste sempre un modo di eliminare la ricorsione, ma può essere molto complesso
  - La ricorsione è semplice ed elegante (anche se lenta)