Corso di Laurea Ingegneria Informatica
Fondamenti di Informatica 2

Dispensa 04

Ricorsione

A. Miola

Febbraio 2008

Contenuti

- Funzioni e domini definiti induttivamente
- Ricorsione e metodi ricorsivi
- Gestione della memoria a run-time
- Ricorsione multipla

Ricorsione

- **La ricorsione è una tecnica di programmazione basata sulla definizione induttiva (o ricorsiva) di funzioni su domini che sono definiti in modo induttivo (o ricorsivo)**
  - **una funzione è definita in modo induttivo se è definita, direttamente o indirettamente, in termini di se stessa – funzione ricorsiva**
  - **un dominio è definito in modo induttivo se è definito, direttamente o indirettamente, in termini di se stesso - dominio ricorsivo**

Esempi di definizioni ricorsive

- La funzione fattoriale può essere definita ricorsivamente con la seguente definizione per casi

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

- L'insieme $\mathbb{N}$ dei numeri naturali può essere definito in modo induttivo come segue
  - $0 \in \mathbb{N}$
    - lo zero è un numero naturale
  - se $n \in \mathbb{N}$, allora anche $n+1 \in \mathbb{N}$
    - il successore di un numero naturale è un numero naturale

Principio di induzione matematica

- La ricorsione è connessa alla nozione di induzione matematica
- La fondatezza della ricorsione può essere dimostrata sulla base del seguente principio di induzione matematica
  - sia P(n) una proposizione definita per n naturale
  - la proposizione P(n) è vera per ogni numero naturale n se
    - P(0) è vera - (*per definizione*) passo base dell'induzione
    - per ogni naturale k, se P(k) è vera (ipotesi induttiva) allora anche P(k+1) è vera - passo induttivo

Ad esempio

- Sia P(n) la proposizione definita per n numero naturale per cui n! = n*(n -1)! = 1*2*3 . . . * n
- La proposizione P(n) è vera per ogni numero naturale n se
  - P(0) è vera - passo base
    - 0! = 1 - (per definizione)
  - per ogni naturale k, se P(k) è vera (ipotesi induttiva) allora anche P(k+1) è vera - passo induttivo
    - se P(k) è vera : k! = k*(k-1)! = 1*2* … *k
    - allora anche P(k+1) è vera –

      (k+1)! = (k+1)*k! = (k+1)*(1*2* … *k) = 1*2* … *k*(k+1)

Domini definiti induttivamente

- **Molto spesso i dati che devono essere manipolati da un programma appartengono ad un dominio definito induttivamente**
- **L'insieme degli elementi di un dominio definito induttivamente può essere caratterizzato da**
  - **uno o più elementi (in numero finito) che appartengono al dominio (per definizione)**
  - **una o più regole, ciascuna delle quali permette di ottenere, a partire da uno o più elementi (che sappiamo essere parte) del dominio, un nuovo elemento del dominio**

Nessun altro elemento, oltre a quelli specificati, appartiene al dominio

Definizioni induttiva dell'insieme dei numeri naturali

- L'insieme $\mathbb{N}$ dei numeri naturali può essere definito in modo induttivo come segue
  - $0 \in \mathbb{N}$
    - lo zero è un numero naturale
  - se $n \in \mathbb{N}$, allora anche $n+1 \in \mathbb{N}$
    - il successore di un numero naturale è un numero naturale

- Esercizio
  - verificare, sulla base della precedente definizione, che 3 è un numero naturale

Definizioni induttiva dell'insieme delle stringhe su un alfabeto

- **L'insieme S_Σ delle stringhe su un alfabeto Σ di caratteri può essere definito in modo induttivo come segue**
  - **"" $\in S_\Sigma$**
    - **la stringa vuota è una stringa su Σ**
  - **se c $\in \Sigma$ e S $\in S_\Sigma$, allora anche c+S $\in S_\Sigma$**
    - **un carattere in Σ concatenato a una stringa su Σ è una stringa su Σ**

- **Esercizio**
  - **verificare, sulla base della precedente definizione, che "ab" è una stringa (sull'alfabeto Unicode)**

Definizioni induttiva dell’insieme dei file di testo

- **L’insieme dei file di testo può essere definito in modo induttivo come segue**
  - il file vuoto è un file di testo
  - se *f* è un file di testo, allora mettendo in testa ad *f* una nuova stringa di caratteri si ottiene un file di testo

Definizione induttiva di un insieme

- **In generale quindi nella definizione induttiva di un insieme è possibile identificare**
  - **uno o più casi base**
    - **un caso base descrive l'appartenenza di alcuni elementi all'insieme in modo diretto**
  - **uno o più casi induttivi**
    - **un caso induttivo descrive l'appartenenza di alcuni elementi all'insieme in modo indiretto, ovvero in termini dell'appartenenza di altri elementi all'insieme stesso**

Definizione ricorsiva di una funzione

- **La definizione ricorsiva di una funzione ricalca la struttura della definizione induttiva del dominio su cui opera la funzione, per cui abbiamo**
  - **uno (o più) casi base, per i quali il risultato può essere determinato direttamente**
  - **uno (o più) casi ricorsivi, per i quali si riconduce il calcolo del risultato al calcolo della stessa funzione su un valore più piccolo (più semplice)**

Fondatezza della definizione ricorsiva di una funzione

- I diversi casi partizionano il dominio della funzione ovvero, sono mutuamente esclusivi e gli elementi da essi definiti coprono il dominio
- Il fatto che il dominio su cui opera la funzione sia definito induttivamente, ci garantisce che, applicando ripetutamente i casi ricorsivi, ci riconduciamo prima o poi (e sempre in un numero finito di passi - cioè si ha la terminazione) ad uno dei casi base

Definizione ricorsiva della funzione fattoriale

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 \text{ – caso base} \\ n \cdot fatt(n-1) & \text{se } n > 0 \text{ – caso induttivo} \end{cases}$$

- La definizione ricorsiva della funzione **fattoriale** può essere implementata direttamente in Java, attraverso un **metodo ricorsivo**

```
public static long fattoriale(long n) {
        long f;
        if (n == 0) f = 1;
        else f = n * fattoriale(n - 1);
        return f;
}
```

Implementazione ricorsiva della somma di due numeri interi

- Definizione ricorsiva funzione somma tra 2 interi non negativi:

$$somma(x,y) = \begin{cases} x & \text{se } y = 0 \text{ - caso base} \\ 1 + somma(x, y-1) & \text{se } y > 0 \text{ - caso induttivo} \end{cases}$$

- Definizione ricorsiva della funziona somma in Java:

```
public static int somma(int x, int y) {
    int s;
    if (y == 0) s = x;              // se y = 0
    else s = 1 + somma(x, y-1);     // se y > 0
    return s;
}
```

Implementazione ricorsiva del prodotto di due numeri interi

- Definizione ricorsiva funzione prodotto tra 2 interi non negativi:

$$prodotto(x,y) = \begin{cases} 0 & \text{se } y = 0 \\ somma(x, prodotto(x, y-1)) & \text{se } y > 0 \end{cases}$$

- Implementazione della funzione prodotto, supponendo che il metodo somma sia definito nella stessa classe:

```
public static int prodotto(int x, int y) {
    int p;
    if (y == 0) p = 0;                       // se y = 0
    else p = somma(x, prodotto(x, y-1));  // se y > 0
     return p;
 }
```

Implementazione ricorsiva dell’elevamento a potenza

- Definizione ricorsiva funzione potenza tra 2 interi non negativi:

$$potenza(b,e) = \begin{cases} 1 & \text{se } e = 0 \\ prodotto(b, potenza(b, e-1)) & \text{se } e > 0 \end{cases}$$

- Implementazione, supponendo che il metodo prodotto (e quindi anche il metodo somma) sia definito nella stessa classe:

```
public static int potenza(int b, int e) {
  int p;
  if (e == 0) p = 1;                          // se e = 0
  else p = prodotto(b, potenza(b, e-1));  // se e > 0
  return p;
 }
```

Confronto tra ricorsione e iterazione . . .

- Alcuni metodi implementati in modo ricorsivo ammettono anche un'implementazione iterativa (non-ricorsiva), ad esempio sviluppando il fattoriale di n:

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

  possiamo scrivere un metodo che effettua una iterazione per calcolare i prodotti a partire dal parametro n fino a 1, senza che ci siano chiamate alla funzione *fatt* all'interno del metodo stesso

. . . Confronto tra ricorsione e iterazione

- Implementazione iterativa della funzione fattoriale :

```
public static long fattorialeIterativa (long n) {
  long ris = 1;          // inizializzazione
  while (n > 0) {        // ciclo da n a 1
    ris = ris * n;
    n--;
  }
  return ris;
}
```

- Esercizio
  - Dimostrare che le implementazioni ricorsiva e iterativa sono equivalenti

Numero di occorrenze di un carattere in una stringa . . .

- Formalizzazione ricorsiva della funzione che conta le occorrenze del carattere c nella stringa s:
  - se s è la stringa vuota “”, restituisci 0 – passo base
  - altrimenti – passo induttivo
    - se il primo carattere di s è uguale a c, restituisci 1 più il numero di occorrenze di c nella stringa data da s senza il primo carattere
    - altrimenti (ovvero se il primo carattere di s è diverso da c), allora restituisci il numero di occorrenze di c nella stringa data da s tranne il primo carattere

. . . Numero di occorrenze di un carattere in una stringa

- **Implementazione ricorsiva:**

```
public static int contaLettera(String s, char c) {
    int cont;
    if (s.length() == 0) cont = 0;
    else if (s.charAt(0) == c)
      cont = 1 + contaLettera(s.substring(1), c);
      else
          cont = contaLettera(s.substring(1), c);
    return cont;
}
```

Stringa palindroma . . .

- **Una stringa si dice palindroma se la stringa coincide con la stringa stessa letta da destra verso sinistra, ad esempio:**

  "itopinonavevanonipoti"

- **Una caratterizzazione induttiva di una stringa palindroma:**
  - **la stringa vuota è palindroma;**
  - **una stringa costituita da un singolo carattere è palindroma**
  - **una stringa *c s d* è palidroma, se *s* è una stringa palidroma e *c* ed *d* sono due caratteri uguali;**

. . . Stringa palindroma

- Possibile implementazione ricorsiva:

```
public static boolean palindroma(String s) {
  boolean ret;
  if (s.length() <= 1)
    ret = true;
  else
    ret = (s.charAt(0)==s.charAt(s.length()-1))
     && palindroma(s.substring(1,s.length()-1));
  return ret;
}
```

Gestione della memoria a run-time

A tempo di esecuzione, la macchina virtuale Java (JVM) deve gestire diverse zone di memoria :

- **zona codice che contiene il Java bytecode (ovvero il codice eseguibile dalla JVM)**
  - **determinata a tempo di esecuzione al momento del caricamento della classe**
  - **dimensione fissata per ogni metodo a tempo di compilazione**
- **heap (o mucchio): zona di memoria che contiene gli oggetti**
  - **cresce e descresce dinamicamente durante l'esecuzione**
  - **ogni oggetto viene allocato e deallocato independentemente dagli altri**
- **pila dei record di attivazione (o stack): zona di memoria per i dati locali ai metodi (variabili e parametri)**
  - **cresce e descresce dinamicamente durante l'esecuzione**
  - **viene gestita con un meccanismo a pila**

Gestione dello heap e garbage collection

- **Un oggetto viene creato invocando un costruttore tramite l'operatore new**
  - **Al momento della creazione di un oggetto, la zona di memoria per l'oggetto stesso viene riservata nello heap**
- **Quando un oggetto non è più utilizzato da un programma, la zona di memoria allocata nello heap per l'oggetto può essere liberata e resa disponibile per nuovi oggetti**
  - **In Java, a differenza di altri linguaggi, il programmatore non può effettuare tale operazione esplicitamente. Essa viene effettuata automaticamente dal garbage collector, quando l'oggetto non è più accessibile**

Gestione dello heap e garbage collection

- Il garbage collector è una componente della JVM che è in grado di rilevare quando un oggetto non ha più riferimenti ad esso, e quindi di fatto non è più utilizzabile e può essere deallocato
- Tipicamente, il garbage collector viene invocato automaticamente dalla JVM, senza controllo da parte del programmatore, quando si rende necessario rendere disponibile della memoria

Pila dei record di attivazione

- Una pila (o stack) è una struttura dati con accesso LIFO: Last In First Out
- A run-time la JVM gestisce la pila dei Record di Attivazione (RDA):
  - per ogni attivazione di metodo viene creato un nuovo RDA in cima alla pila
  - al termine dell'attivazione del metodo il RDA viene "rimosso" dalla pila

Record di attivazione

Un Record di attivazione - RDA - contiene :

- le locazioni di memoria per i parametri formali, incluso l'eventuale riferimento all'oggetto di invocazione (se presenti)
- le locazioni di memoria per le variabili locali (se presenti)
- il valore di ritorno dell'invocazione del metodo (se il metodo ha tipo di ritorno diverso da void)
- una locazione di memoria per l'indirizzo di ritorno, ovvero l'indirizzo della successiva istruzione da eseguire nel metodo chiamante

Esempio di evoluzione della pila dei record di attivazione

Esempio: Cosa avviene durante l'esecuzione del main?

```
public static int B(int pb) {
  /* b0 */ System.out.println("In B. Parametro pb = " + pb);
  /* b1 */ return pb+1;
}

public static int A(int pa) {
  /* a0 */ System.out.println("In A. Parametro pa = " + pa);
  /* a1 */ System.out.println("Chiamata di B(" + (pa * 2) + ").");
  /* a2 */ int va = B(pa * 2);
  /* a3 */ System.out.println("Di nuovo in A. va = " + va);
  /* a4 */ return va + pa;
}

public static void main(String[] args) {
  /* m0 */ System.out.println("In main.");
  /* m1 */ int vm = 22;
  /* m2 */ System.out.println("Chiamata di A(" + vm + ").");
  /* m3 */ vm = A(vm);
  /* m4 */ System.out.println("Di nuovo in main. vm = " + vm);
  /* m5 */ return;
}
```

Evoluzione della pila di attivazione . . .

- **Per semplicità**
  - ignoriamo l'invocazione println, trattandola come se fosse una istruzione elementare
  - supponiamo che ad ogni istruzione del codice sorgente Java corrisponda una singola istruzione in Java bytecode
  - assumiamo che il bytecode venga caricato dalla JVM nelle seguenti locazioni di memoria:

```
          main                        A                          B
       |  ...   |               |  ...   |                  |  ..    |
   100 |  m0    |           200 |  a0    |              300 |  b0    |
   101 |  m1    |           201 |  a1    |              301 | return |
   102 |  m2    |           202 |  a2    | ⇒ B(va*2)    302 |  ..    |
   103 |  m3    | ⇒ A(vm)   203 |  a3    |
   104 |  m4    |           204 | return |
   105 | return |           205 |  ..    |
   106 |  ..    |
```

. . . Evoluzione della pila di attivazione . . .

- Output prodotto dal programma:

```
In main
Chiamata di A(22).
In A. Parametro pa = 22
Chiamata di B(44).
In B. Parametro pb = 44
Di nuovo in A. va = 45
Di nuovo in main. vm = 67
```

- Per comprendere cosa avviene durante l'esecuzione del codice, è necessario fare riferimento, oltre che alla pila dei RDA, al Program Counter (PC) cioè l'indirizzo della prossima istruzione da eseguire

. . . Evoluzione della pila di attivazione . . .

- Analizziamo in dettaglio cosa avviene al momento dell'attivazione di A(vm) dal metodo main
  1. vengono valutati i parametri attuali: nel nostro caso il parametro attuale è l'espressione vm che ha come valore l'intero 22

  2. viene individuato il metodo da eseguire in base al numero e tipo dei parametri attuali, cercando la definizione di un metodo la cui segnatura sia conforme alla invocazione: nel nostro caso il metodo da eseguire deve avere la segnatura A(int pa)
  3. viene sospesa l'esecuzione del metodo chiamante: nel nostro caso il metodo main

. . . Evoluzione della pila di attivazione . . .

4. viene creato il RDA relativo all'attivazione corrente del metodo chiamato: nel nostro caso viene creato il RDA relativo all'attivazione corrente di A; il RDA contiene:

- le locazioni di memoria per i parametri formali: nel nostro caso, il parametro pa di tipo int
- le locazioni di memoria per le variabili locali: nel nostro caso, la variabile va di tipo int
- una locazione di memoria per il valore di ritorno: nel nostro caso indicata con VR
- una locazione di memoria per l'indirizzo di ritorno: nel nostro caso indicata con IR

5. viene assegnato il valore dei parametri attuali ai parametri formali: nel nostro caso, il parametro formale pa viene inizializzato con il valore 22

. . . Evoluzione della pila di attivazione . . .

6. l'indirizzo di ritorno nel RDA viene impostato all'indirizzo della successiva istruzione che deve essere eseguita nel metodo chiamante al termine dell'invocazione:

- nel nostro caso, l'indirizzo di ritorno nel RDA relativo all'attivazione di *A* viene impostato al valore *104*, che è l'indirizzo dell'istruzione di *main* corrispondente all'istruzione *m4*, da eseguire quando l'attivazione di *A* sarà terminata; a questo punto, la pila dei RDA è come segue:

A	va	?
	pa	22
	VR	?
	IR	104
main	vm	22

. . . Evoluzione della pila di attivazione . . .

7. al PC viene assegnato l'indirizzo della prima istruzione del metodo invocato: nel nostro caso, al program counter viene assegnato l'indirizzo *200*, che è l'indirizzo della prima istruzione di *A*

8. si passa ad eseguire la prossima istruzione indicata dal PC, che sarà la prima istruzione del metodo invocato: nel nostro caso l'istruzione di indirizzo *200*, ovvero la prima istruzione di *A*

. . . Evoluzione della pila di attivazione . . .

- Dopo questi passi, le istruzioni del metodo chiamato, (cioè *A*), vengono eseguite in sequenza. In particolare, avverrà l'attivazione, l'esecuzione e la terminazione di eventuali metodi a loro volta invocati nel metodo chiamato.
- Nel nostro caso, avverrà l'attivazione l'esecuzione e la terminazione del metodo *B*, con un meccanismo analogo a quello adottato per *A*

	pb	44
B	VR	45
	IR	203
	va	?
A	pa	22
	VR	?
	IR	104
main	vm	22

. . . Evoluzione della pila di attivazione . . .

- **Analizziamo cosa avviene al momento della terminazione dell'attivazione di *A*, ovvero quando viene eseguita l'istruzione return va+pa:**
- **Prima di questa esecuzione, la pila dei RDA è la seguente:**

A	va	45
	pa	22
	VR	67
	IR	104
main	vm	22

- **Nota: in realtà, la zona di memoria predisposta a contenere il valore di ritorno, (*VR*), viene inizializzata contestualmente all'esecuzione dell'istruzione return, e non prima**

. . . Evoluzione della pila di attivazione . . .

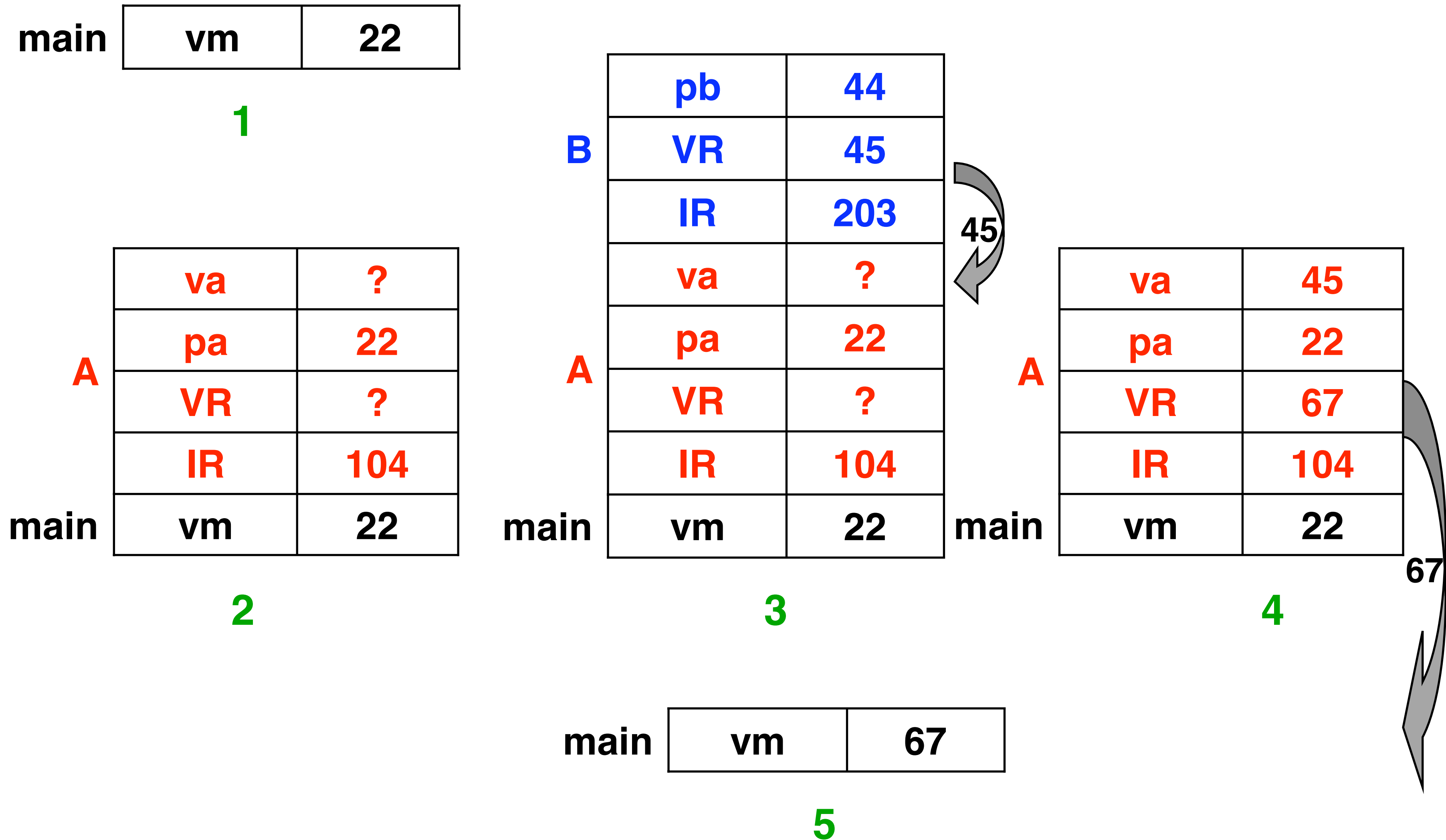
1. al PC viene assegnato il valore memorizzato nella locazione di memoria riservata all'indirizzo di ritorno nel RDA corrente: nel nostro caso, tale valore è pari a *104*, che è proprio l'indirizzo, memorizzato in *IR*, della prossima istruzione di *main* che dovrà essere eseguita;
2. nel caso il metodo invocato preveda la restituzione di un valore di ritorno, tale valore viene memorizzato in un'apposita locazione di memoria del RDA corrente: nel nostro caso, il valore *67*, risultato della valutazione dell'espressione *va+pa* viene assegnato alla locazione di memoria indicata con *VR*

. . . Evoluzione della pila di attivazione . . .

3. viene eliminato dalla pila dei RDA il RDA relativo all'attivazione corrente, e il RDA corrente diviene quello precedente nella pila; contestualmente all'eliminazione del RDA dalla pila, un eventuale valore di ritorno viene copiato in una locazione di memoria del RDA del chiamante: nel nostro caso, viene eliminato il RDA di *A* e il RDA corrente diviene quello relativo all'attivazione di *main*; il valore *67*, memorizzato nella locazione di memoria *VR* viene assegnato a *vm* nel RDA di *main*:

4. si passa ad eseguire la prossima istruzione indicata dal PC , ovvero quella appena impostata al passo 1: nel nostro caso, si passa ad eseguire l'istruzione di indirizzo *104*, che fa riprendere l'esecuzione di *main*

. . . Evoluzione della pila di attivazione



La pila di attivazione per i metodi ricorsivi

- Per le chiamate ricorsive occorre ricordare che un RDA non è associato ad un metodo, ma a una singola attivazione dello stesso – Ad esempio :

```
public static void ricorsivo(int i) {
  System.out.print("In ricorsivo(" + i + ")");
  if (i == 0)
    System.out.println(" - Finito");
  else {
    System.out.println(" - Attivazione di
                ricorsivo(" + (i-1) + ")");
    ricorsivo(i-1);
    System.out.print("Di nuovo in ricorsivo
                       (" + i + ")");
    System.out.println(" - Finito");
  }
  return;
}     . . . segue . . .
```

La pila di attivazione per i metodi ricorsivi

```
    . . . segue . . .
public static void main(String[] args) {
  int j;
  System.out.print("In main");
  j = 2;
  System.out.println("- Attivazione di
          ricorsivo ("+j+ ")");
  ricorsivo(j);
  System.out.print("Di nuovo in main");
  System.out.println(" - Finito");
}
```

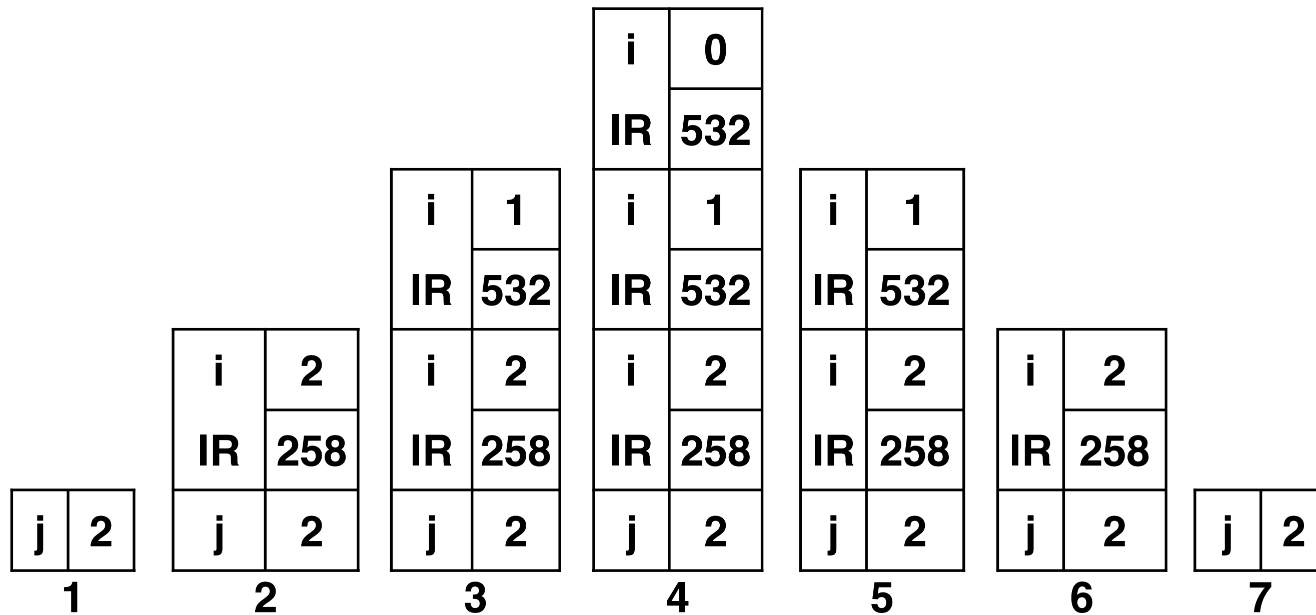
La pila di attivazione per i metodi ricorsivi

- Esempio di Output con input pari a 2:

```
In main - Attivazione di ricorsivo(2)
In ricorsivo(2) - Attivazione di ricorsivo(1)
In ricorsivo(1) - Attivazione di ricorsivo(0)
In ricorsivo(0) - Finito
Di nuovo in ricorsivo(1) - Finito
Di nuovo in ricorsivo(2) - Finito
Di nuovo in main - Finito
```

La pila di attivazione per i metodi ricorsivi

- **Assumendo che *258* sia l'indirizzo dell'istruzione che segue l'attivazione di ricorsivo(j) in main, e che *532* sia l'indirizzo dell'istruzione che segue l'attivazione di ricorsivo(i-1) in ricorsivo abbiamo i seguenti RDA:**



La pila di attivazione per i metodi ricorsivi

- Nella pila non ci sono valori di ritorno;
- Il RDA in fondo alla pila è relativo a main, e tutti gli altri sono relativi ad attivazioni successive di ricorsivo
- Per le diverse attivazioni ricorsive vengono creati diversi RDA sulla pila, con valori via via decrescenti del parametro i fino a 0
- Il bytecode associato alle diverse attivazioni ricorsive è sempre lo stesso, ovvero quello del metodo ricorsivo. Perciò l'indirizzo di ritorno nei RDA per le diverse attivazioni ricorsive è sempre lo stesso (532), tranne che per la prima attivazione, per la quale l'indirizzo di ritorno è quello di un'istruzione nel metodo main (258)

Esercizio

- Descrivere l'esecuzione della seguente applicazione, fornendo le successive configurazioni della pila di attivazione

```
public class Esecuzione {

    public static void main(String[] args) {
            fattoriale(3);
    }

    public static long fattoriale(long n) {
            long f;
            if (n == 0) f = 1;
            else f = n * fattoriale(n - 1);
            return f;
    }
}
```

Ricorsione multipla . . .

- Si ha una ricorsione multipla quando un'attivazione di un metodo può causare più di una attivazione ricorsiva dello stesso metodo
- Esempio: il numero di Fibonacci *n:*

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-2) + F(n-1) & \text{se } n > 1 \end{cases}$$

- Sequenza di Fibonacci: $F(0), F(1), F(2), \ldots$

  cioè: 0, 1, 1, 2, 3, 5, 8, 13, 21

. . . Ricorsione multipla

- Implementazione ricorsiva Fibonacci che prende come paramentro un intero positivo **n**:

```
public static long fibonacci(long n) {
   long ret;
   if (n == 0)
      ret = 0;
   else if (n == 1)
      ret = 1;
   else
      ret = fibonacci(n-1) + fibonacci(n-2);
   return ret;
}
```

Torri di Hanoi . . .

- **Nel problema delle Torri di Hanoi occorre spostare una torre di dischi seguendo le seguenti regole:**
  1. **inizialmente, la torre di dischi di dimensione decrescente è posizionata su un perno 1;**
  2. **l'obiettivo è quello di spostarla su un perno 2, usando un perno 3 di appoggio;**

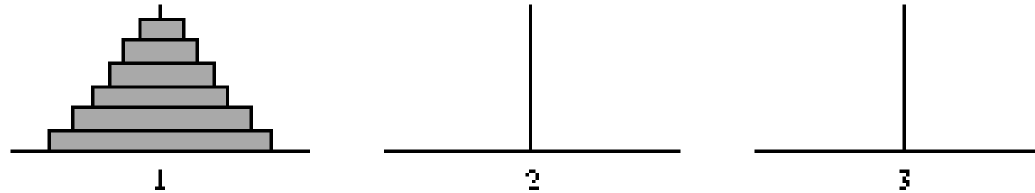
. . . Torri di Hanoi . . .

3. **le condizioni per effettuare gli spostamenti sono:**
   1. tutti i dischi, tranne quello spostato, devono stare su una delle torri
   2. è possibile spostare un solo disco alla volta, dalla cima di una torre alla cima di un'altra torre;
   3. un disco non può mai stare su un disco più piccolo

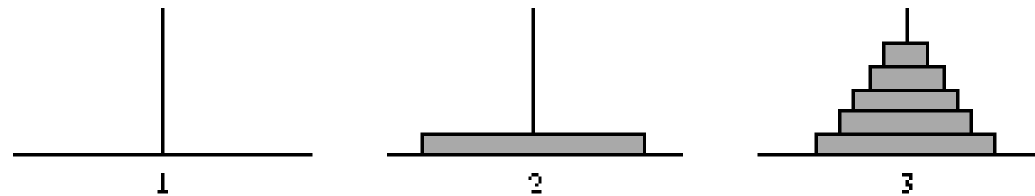
. . . Torri di Hanoi . . .

Esempio di esecuzione:

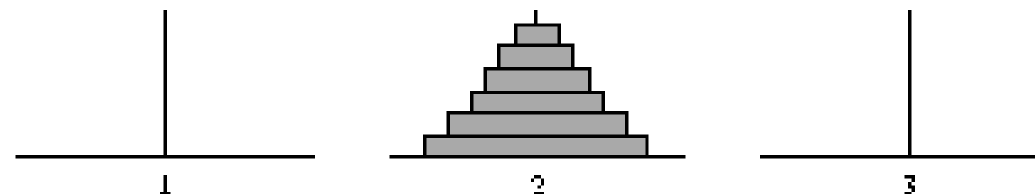
- stato iniziale



- stato intermedio



- stato finale



. . . Torri di Hanoi . . .

- **Per realizzare un programma che stampi la sequenza di spostamenti da fare si può pensare di ragionare ricorsivamente: per spostare *n* dischi (con *n* > 1) da 1 a 2, utilizziamo 3 come appoggio:**
  - **sposta (ricorsivamente) *n* - 1 dischi da 1 a 3**
  - **sposta l'*n*-esimo disco da 1 a 2**
  - **sposta (ricorsivamente) *n* - 1 dischi da 3 a 2**
- **Il problema di spostare *n* dischi diventa quello - più semplice - di spostare *n - 1* dischi**

. . . Torri di Hanoi . . .

- Implementazione ricorsiva per il problema delle Torri di Hanoi:

```
public class Hanoi {
  private static void muoviUnDisco(int sorg, int dest) {
     System.out.println("muovi un disco da " + sorg +
                        " a " + dest);
  }

  private static void muovi(int n, int sorg, int dest, int aux)
   {
        if (n == 1)muoviUnDisco(sorg, dest);
        else {
           muovi(n-1, sorg, aux, dest);
           muoviUnDisco(sorg, dest);
           muovi(n-1, aux, dest, sorg);
         }
    }  . . . segue . . .
```

. . . Torri di Hanoi

```
     . . . segue . . .
 public static void main (String[] args) {

 int n = Lettore.in.leggiInt();
 System.out.println("Per muovere " + n +
              " dischi da 1 a 2 con 3 come appoggio:");
 muovi(n, 1, 2, 3);

 System.exit(0);
 }
}
```

Numero di attivazioni nel caso di metodi ricorsivi

- La JVM assegna alla Pila dei RDA un numero fissato di locazioni di memoria
- Quando si usa la ricorsione bisogna quindi tener presente il numero di attivazioni ricorsive che potrebbe essere tale da esaurire la disponibilità di memoria massima assegnata alla pila dei RDA
- Si rifletta ad esempio al caso del calcolo ricorsivo del fattoriale di 10, di 100 o di 1000

Numero di attivazioni nel caso di ricorsione multipla . . .

- Quando si usa la ricorsione multipla, bisogna tenere presente che il numero di attivazioni ricorsive potrebbe essere esponenziale nella profondità delle chiamate ricorsive, cioè nell'altezza massima della pila dei RDA
- Nel caso di Hanoi, il numero di attivazioni di muoviUnDisco (cioè di spostamenti di dischi) per n dischi è:

$$att(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1 + 2 \cdot att(n-1) & \text{se } n > 1 \end{cases}$$

. . . Numero di attivazioni nel caso di ricorsione multipla

- Nel caso $n > 1$, a meno del fattore 1, si ha:

$$att(n) > 2^{n-1}$$

- Nota: nel caso del problema delle Torri di Hanoi il numero esponenziale di attivazioni è una caratteristica intrinseca del problema, nel senso che non esiste una soluzione migliore

Riferimenti

- Per lo studio di questi argomenti si fa riferimento al libro di testo, e in particolare ai Capitoli 21 e 22