

DIPARTIMENTO DI INFORMATICA – TORINO

PROGRAMMAZIONE 1 — DISPENSE A.A. 2019/2020¹

Roversi, L. and Cardone, F.

12 dicembre 2019

¹Ogni parte di questo **Work** per la quale non sia altrimenti specificato, sia essa fruibile in formato cartaceo o per mezzo di un ausilio elettronico, e sviluppata dall'**Original author**, e solo quelle, sono distribuite in accordo con la licenza [Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported](#).

Indice

I In AULA	7
1 Pensiero computazionale e algoritmi	9
1.1 Il “posto” della programmazione tra le capacità di un informatico	9
1.2 Pensiero computazionale e vita quotidiana	9
2 Programmazione strutturata iterativa di base	17
2.1 Assegnazione	17
2.1.1 Assegnazione e configurazioni	17
2.1.2 Assegnazione e sequenza	18
2.2 Selezione e azioni condizionate	19
2.3 Ripetizione di azioni e costrutti iterativi	20
2.3.1 Quadrato di un numero intero basato sui prodotti notevoli	23
2.3.2 Iterazioni di iterazioni	25
2.3.3 Riferimenti bibliografici	26
3 Metodi e Modello di gestione della memoria	27
3.1 <i>Frame</i> e variabili locali al main	27
3.2 Metodi senza risultato, ma con parametri	29
3.3 Metodi con parametri e risultato	33
3.4 Campi statici e final	37
3.5 <i>Signature, overloading, cast, etc.</i>	41
3.6 Jeliot	45
4 Elementi di base per la Correttezza parziale	47
4.1 Correttezza parziale all’opera	48
4.1.1 Introduzione dell’invariante di ciclo	49
4.2 Dimostrare che un predicato è invariante	50
4.2.1 Invariante di SIDP	50
4.2.2 Correttezza parziale per QPNP	56
4.2.3 Predicati per selezioni	57
4.2.4 Invariante per il problema MCD	58
4.3 Il principio di induzione	59
4.4 Dimostrazione della Correttezza parziale per induzione	61
4.4.1 Correttezza parziale di SIDP	61
4.4.2 Correttezza parziale di QPNP	67
4.4.3 Correttezza parziale di QRDIP	67
4.4.4 Correttezza parziale per un algoritmo di calcolo del quadrato	69
4.5 Correttezza parziale con predicati implicativi	70
4.6 Correttezza parziale per SNNP	71
4.6.1 Iterazioni annidate	73
4.6.2 Iterazioni annidate e correttezza senza predicati implicativi	76
4.6.3 Approfondimenti facoltativi	76
5 Programmazione ricorsiva di base	77
5.1 Definizioni e computazioni ricorsive	77
5.1.1 Fattoriale	77
5.1.2 Quadrato	80
5.2 Ricorsione e correttezza parziale per induzione	81
5.2.1 Sommatoria di un segmento di naturali	82
5.2.2 Lettura e stampa di una sequenza di valori	83

5.2.3 Ricorsione “controvariante” e metodi involucro	84
5.3 Ricorsione di coda	86
5.4 Induzione forte e funzioni ricorsive <i>parallelizzabili</i>	88
5.4.1 Quoziente tra naturali per sottrazioni iterate	88
5.4.2 Resto del quoziente tra naturali per sottrazioni iterate	89
5.4.3 Funzione identità	90
5.4.4 Funzione successore dicotomica	90
5.5 Funzioni ricorsive famose (NON nel programma didattico)	92
5.5.1 Coefficiente binomiale	92
5.5.2 Torre di Hanoi	92
5.5.3 Funzione di McCarty	93
5.5.4 Funzione di Ackermann	93
6 Programmazione con array	95
6.1 Una “scusa” per introdurre gli <i>array</i>	95
6.2 <i>Array</i> e gestione della memoria: <i>Heap</i>	96
6.3 Creazione di <i>array</i> , inizializzazione ed egualanza	98
6.3.1 Perché gli <i>array</i> sono nella <i>heap</i> ?	100
6.4 Eguaglianza tra <i>array</i> e <i>aliasing</i>	101
6.4.1 Classe <i>Array</i>	103
6.5 Operazioni di base su <i>array</i>	104
6.5.1 Ricerca lineare	104
6.5.2 Filtri	105
6.5.3 Inserimenti e cancellazioni	105
6.6 <i>Array</i> e visite dicotomiche o parallelizzabili	105
6.6.1 Schema dicotomico generale	105
6.7 Matrici bidimensionali	108
6.7.1 Problemi decisionali	109
6.7.2 Ricerca dicotomica su <i>array</i> ordinati	111
6.8 Ordinamenti ed operazioni su <i>array</i> ordinati (NON è parte del programma didattico)	111
6.8.1 Ordinamenti come visita di uno spazio di configurazioni	111
6.8.2 Bubble sort	113
6.8.3 Selection sort	114
6.8.4 Insertion sort	114
6.8.5 Fusione di due <i>array</i> ordinati	115
6.8.6 Merge sort	115
6.8.7 Costi di algoritmi di ordinamento	115
6.8.8 Eventuali approfondimenti su Algoritmi di ordinamento (per curiosi)	115
6.9 Approfondimenti eventuali su <i>Array</i> (NON è parte del programma didattico)	116
6.9.1 Iterazione e ricorsione su <i>array</i>	116
6.9.2 Strutture dati con <i>array</i>	118
6.9.3 Indirizzamento indiretto: Crivello di Eratostene, Counting sort	119
II In LABORATORIO	121
1 Primi passi in laboratorio	123
1.1 Il PC da linea di comando	123
1.1.1 Struttura del <i>File system</i> in forma testuale	124
1.1.2 Strumenti per la codifica	127
1.1.3 NotePad++	128
1.2 Compilazione e interpretazione (a.a. 17/18 leggere)	130
1.2.1 Struttura essenziale di un PC	130
1.2.2 Compilare un sorgente Java ed interpretare il suo “oggetto”	130
1.2.3 Interpretazione di <i>Espressione.class</i> a livello di CPU	133
1.2.4 Letture integrative	138

2 Il linguaggio JAVA	139
2.1 Utilizzo essenziale delle classi	139
2.1.1 Codifica di primi algoritmi in sorgenti Java	139
2.2 Tipi di base	140
2.3 Ambiguità sintattica della selezione	140
2.4 Operatori ed espressioni Java	140
2.5 Programmazione e codifica: stili	140
2.6 Sviluppo di codice con classi e metodi	140
2.6.1 Classe con metodi iterativi per l'aritmetica	141
2.7 Input da tastiera	141
2.8 Tipi numerici non interi	142
2.9 Operazioni su <i>array</i>	142
2.10 Il tipo <i>char</i>	143
2.11 La classe <i>String</i>	143
2.12 Array multidimensionali	144
2.13 Problemi decisionali su <i>array</i>	145
2.14 Eventuali approfondimenti (NON parte del programma didattico)	145
2.14.1 API Java	145
2.14.2 <i>Package</i>	145
2.15 <i>Assert</i> in Java	146

Parte I

In AULA

Capitolo 1

Pensiero computazionale e algoritmi

1.1 Il “posto” della programmazione tra le capacità di un informatico

Saper programmare un computer è solo una delle componenti il sapere di un informatico. In particolare, programmare è uno dei mattoni che dovrebbero costituire quel che sempre più insistentemente viene chiamato “pensiero computazionale” (*computational thinking*).

A prescindere da mode e terminologie che, nel tempo, possono raffinare e meglio identificare soggetti e ambiti di cui intendono trattare, con “pensiero computazionale” si parla di un insieme di capacità e strumenti che, se di patrimonio comune, possono incrementare comprensione e gestione della complessa società in cui viviamo.

In “[Computational Thinking](#)” Janette Wing sviscera, usando diverse prospettive, il concetto di *computational thinking*, riferendosi esplicitamente al posto che la programmazione debba occupare nel bagaglio culturale informatico:

«*Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.*»

La programmazione, quindi, deve essere un mattone che, tuttavia, non costituisce l’intera casa (culturale).

La spiegazione di cosa è il pensiero computazionale, fornita da Wing, sfrutta parole che difficilmente appartengono al gergo comune. Il primo paragrafo è emblematico. Una traduzione letterale ragionevolmente fedele, che tiene conto dello scopo di questa introduzione, è la seguente:

«Il pensiero computazionale si basa sulle capacità e le limitazioni dei processi computazionali, indipendentemente dal fatto che essi siano eseguiti da un umano o da un computer. I metodi ed i modelli computazionali ci danno il coraggio di risolvere problemi e progettare sistemi che nessuno di noi sarebbe in grado di affrontare da solo. ... A livello più fondamentale [definire cosa significhi “pensiero computazionale”] equivale alla questione: “Cosa è computabile?”. Oggi conosciamo solo parti delle risposte a tale domanda.»

Il paragrafo cita disinvoltamente “processo”, “metodi e modelli” computazionali e ricorda che alla domanda “Cosa è computabile” non sappiamo rispondere efficacemente.

Questo corso di programmazione è impostato per cominciare a capire cosa si può intendere quando si parla di processi computazionali. Tratteremo di processi computazionali attraverso semplici algoritmi, tradotti in un opportuno linguaggio di programmazione che può essere interpretato da un calcolatore che usa opportunamente lo spazio di memoria disponibile e, al termine, produce il risultato voluto. Un “processo computazionale” quindi, richiede l’esplicazione ed il controllo di innumerevoli concetti e tecniche, a diversi livelli di astrazione, che la programmazione rende evidenti al loro livello di base.

Lo sviluppo e la comprensione di altri aspetti del *computational thinking* come, pescando a caso da Wing: la comprensione del comportamento umano, l’uso di astrazione e decomposizione per attaccare problemi difficili, l’adottare euristiche per risolvere un problema, etc. saranno soggetto di ulteriori corsi.

Il nostro scopo è comprendere cosa siano, come siano esprimibili e quale scopo abbiano algoritmi che debbano essere interpretati da calcolatori.

1.2 Pensiero computazionale e vita quotidiana

Per avvicinarci allo scopo appena dichiarato, usiamo esempi ed esercizi il cui scopo è:

Strutturare la descrizione di semplici attività quotidiane per mezzo di costrutti linguistici opportuni.

Esempio 1 (Chiamata telefonica: descrizione strutturata) Telefonare a qualcuno è un'attività che richiede diverse azioni. Se l'obiettivo è descrivere come una telefonata prende corpo occorre rendersi immediatamente conto di quale sia il livello di descrizione che vogliamo raggiungere o, equivalentemente, di quale sia l'interlocutore che debba comprendere la nostra descrizione.

Una delle descrizioni meno impegnative può consistere nella *sequenza* di passi seguente, che identifichiamo come **prima versione**:

1. comporre numero
2. attendere risposta.

Per quanto banale e poco dettagliata, la descrizione già contiene un aspetto rilevante: abbiamo identificato *macro attività* e le abbiamo messe in *sequenza*.

Una seconda osservazione rilevante è che l'eventuale interlocutore verso cui indirizziamo la descrizione deve essere molto sofisticato: in *soli* due punti di descrizione stiamo assumendo che l'interlocutore può capire cosa intendiamo con "comporre" e "attendere", ad esempio.

Proviamo, quindi, ad abbassare le nostre pretese sulle conoscenze dell'interlocutore ed entriamo più nel dettaglio della macro azione "comporre numero". Essa è *macro* perché descrivibile per mezzo di più passi non ancora esplicitati. Lo stesso commento vale per "attendere risposta".

Assumendo la necessità di comporre un numero telefonico di dieci cifre, "comporre numero" può diventare:

- 1.1 inserire cifra;
- 1.2 inserire cifra;
- ⋮ ⋮
- 1.9 inserire cifra;
- 1.10 inserire cifra.

Ancora una volta abbiamo messo in *sequenza* azioni che, per aderire ad un criterio di ragionevolezza a proposito del dettaglio con cui descrivere una telefonata, riteniamo "atomiche": ovvero, d'ora in poi, immaginiamo che "inserire cifra" sia un'operazione che il nostro interlocutore sappia portare a termine senza ulteriori spiegazioni.

Conseguentemente all'aver dettagliato "comporre numero", la **prima versione** è ristrutturabile nella sua **seconda versione**:

1. comporre numero:
 - 1.1 inserire cifra;
 - 1.2 inserire cifra;
 - ⋮ ⋮
 - 1.9 inserire cifra;
 - 1.10 inserire cifra.
2. attendere risposta.

Riflettendo qualche istante, l'*assunzione* su una specifica lunghezza, in questo caso dieci, del numero telefonico da comporre è arbitraria. Se il numero fosse di otto cifre, avremmo necessariamente bisogno di costruire una descrizione di "telefonata" che differisce da quella in via di definizione per il solo numero di cifre?

La soluzione sta nell'adottare uno schema *iterativo* di descrizione come il seguente:

- 1.1 finché(esistono cifre da immettere):
 - 1.1.1 inserire cifra.

Globalmente, la **seconda versione** è ristrutturabile nella **terza versione**:

1. comporre numero:
 - 1.1 finché(esistono cifre da immettere)
 - 1.1.1 inserire cifra.
2. attendere risposta.

Possiamo affermare che la **terza versione** è *più astratta* della seconda perché *indipendente da una scelta a priori del numero di cifre da comporre*.

Ipotizziamo ora che il dettaglio del punto 1 sia quello desiderato: non siamo più interessati ad espanderne la descrizione perché assumiamo che il nostro interlocutore, o *interprete*, sia in grado di comprendere appieno la descrizione data.

In confronto al punto 1, però, il dettaglio del punto 2 è ancora chiaramente grezzo. Per "attendere risposta" è possibile immaginare vari comportamenti. Almeno un paio possono essere evidenziati, chiedendoci cosa può succedere quando si attende la risposta. Ad esempio, fino a che punto attendere in caso di 'numero libero'? Fino alla risposta del chiamato? Per un massimo di 4 squilli o finché non riceviamo segnale "occupato"? E se parte la segreteria telefonica?

Per il solo fatto che esistono, vale la pena d'evidenziare i vari casi e descrivere le azioni corrispondenti a gestirli. Produciamo una **quarta versione** che evidenzia quali delle opzioni possibili siano gestite:

1. comporre numero:
 - 1.1 finché(esistono cifre da immettere)
 - 1.1.1 inserire cifra.
2. attendere risposta:
 - 2.1 se(libero):
 - 2.1.1 finché(gli squilli sono meno di quattro)
 - 2.1.1.1 se(risposta)
 - 2.1.1.1.1 parlare;
 - altrimenti:
 - 2.1.1.1.2 attendere ulteriore squillo;
 - 2.2 chiudere.

Abbiamo deliberatamente tralasciato di gestire l'eventuale interazione con una segreteria telefonica. ■

Esercizio 1 (Telefonata con interazione segreteria) Modificare la **quarta versione** della descrizione di una telefonata dell'Esempio 1, in modo da tenere conto della possibile interazione con una segreteria telefonica. Molto probabilmente, si tratta di modificare il passo 2. ■

Esempio 2 (Attività settimanale studente: descrizione strutturata) Supponiamo di voler strutturare, in analogia con quanto fatto nell'Esempio 1, la descrizione dell'attività semestrale di uno studente universitario con settimane organizzate come segue:

Ora	Lun	Mar	Mer	Gio	Ven
9-10	Prog I B (Aula B)		Log B (Aula B)	Prog I B T2 (Laboratorio Turing)	Mate Discr B (Aula B)
10-11	Prog I B (Aula B)		Log B (Aula B)	Prog I B T2 (Laboratorio Turing)	Mate Discr B (Aula B)
11-12	Mate Discr B (Aula B)	Log B (Aula B)	RO B (Aula B)	Prog I B T2 (Laboratorio Turing)	Prog I B (Aula B)
12-13	Mate Discr B (Aula B)	Log B (Aula B)	RO B (Aula B)		Prog I B (Aula B)
13-14				Mate Discr B (Aula B)	
14-15	RO B (Aula B)	Prog I B T1 (Laboratorio Turing)	Ingl I (Aula A)	Mate Discr B (Aula B)	
15-16	RO B (Aula B)	Prog I B T1 (Laboratorio Turing)	Ingl I (Aula A)		
16-17		Prog I B T1 (Laboratorio Turing)			

Stiamo parlando di descrivere una sequenza di attività. La **prima versione** potrebbe essere banalmente una *sequenza* di macro operazioni:

1. settimana 1
2. settimana 2
- ⋮ ⋮
- m. settimana m
- ⋮ ⋮
- n. settimana ultima,

ciascuna in grado di descrivere il da farsi per ogni settimana.

Supponiamo che, per ora, la prima versione sia accettabile. Il secondo passo è analizzare il da farsi settimanalmente, cosa che dipende dal giorno e dall'ora. Per una settimana generica di indice i, siccome la tabella oraria mostrata non muta, possiamo descrivere la struttura delle attività settimanali in una *sequenza* di possibili *alternative* che dipendono (almeno) dal giorno in cui siamo:

- i. settimana i:
 - i.1 se(è lunedì)
 - i.1.1 attività lunedì;
 - i.2 se(è martedì)
 - i.2.1 attività martedì;
 - i.3 se(è mercoledì)
 - i.3.1 attività mercoledì;
 - i.4 se(è giovedì)
 - i.4.1 attività giovedì;
 - i.5 se(è venerdì)
 - i.5.1 attività venerdì;
 - i.6 se(è sabato)
 - i.6.1 attività sabato;
 - i.7 se(è domenica)
 - i.7.1 attività domenica;

Espandendo la sequenza che costituisce la prima versione della nostra descrizione, otteniamo un elenco di operazioni chiaramente ridondante:

- 1. settimana 1
 - 1.1 se(è lunedì)
 - 1.1.1 attività lunedì;
 - 1.2 se(è martedì)
 - 1.2.1 attività martedì;
 - 1.3 se(è mercoledì)
 - 1.3.1 attività mercoledì;
 - 1.4 se(è giovedì)
 - 1.4.1 attività giovedì;
 - 1.5 se(è venerdì)
 - 1.5.1 attività venerdì;
 - 1.6 se(è sabato)
 - 1.6.1 attività sabato;
 - 1.7 se(è domenica)
 - 1.7.1 attività domenica;
- 2. settimana 2
 - 2.1 se(è lunedì)
 - 2.1.1 attività lunedì;
 - 2.2 se(è martedì)
 - 2.2.1 attività martedì;
 - 2.3 se(è mercoledì)
 - 2.3.1 attività mercoledì;
 - 2.4 se(è giovedì)
 - 2.4.1 attività giovedì;
 - 2.5 se(è venerdì)
 - 2.5.1 attività venerdì;
 - 2.6 se(è sabato)
 - 2.6.1 attività sabato;
 - 2.7 se(è domenica)
 - 2.7.1 attività domenica;
- : : :

Per “ridondante” intendiamo che la descrizione di quel che succede può essere resa molto più concisa, catturando la ripetitività dello schema. Infatti, finché la settimana in cui ci si trova fa parte del primo semestre, dobbiamo ripetere lo schema di comportamento settimanale. Detto equivalentemente, il modo per rendere essenziale la descrizione è *iterare* l’attività settimanale per il numero necessario di volte, ottenendo la **seconda versione** della descrizione che stiamo cercando:

```

finché(settimana è nel semestre)
1 se(è lunedì)
  1.1 attività lunedì;
2 se(è martedì)
  2.1 attività martedì;
3 se(è mercoledì)
  3.1 attività mercoledì;
4 se(è giovedì)
  4.1 attività giovedì;
5 se(è venerdì)
  5.1 attività venerdì;
6 se(è sabato)
  6.1 attività sabato;
7 se(è domenica)
  7.1 attività domenica.

```

Possiamo ora dettagliare la struttura delle attività di qualcuno dei giorni della settimana. L'obiettivo è continuare a seguire l'impostazione data sinora. Si tratta di individuare sequenze di operazioni, eventualmente da iterare sino al raggiungimento di una qualche condizione o tra le quali scegliere per mezzo di qualche criterio.

Focalizziamoci sul martedì. Semplicemente leggendo la tabella, una possibile strutturazione delle attività è la seguente:

attività martedì:

```

1 finché(sono tra le 9 e le 11)
  1.1 se(c'è esercitazione)
    1.1.1 lezione in aula B;
    altrimenti
      1.1.2 studio;
2 finché(sono tra le 11 e le 13)
  1.1 lezione in aula B
3 finché(sono tra le 14 e le 17)
  3.1 se(sono del turno T1)
    3.1.1 lezione in laboratorio Turing;
    altrimenti
      3.1.2 studio.

```

Completato il martedì, siccome il giovedì è simile, pur non prevedendo la possibilità di esercitazioni, possiamo dettagliarne la struttura come segue:

attività giovedì:

```

1 finché(sono tra le 9 e le 12)
  1.1 se(sono del turno T2)
    1.1.1 lezione in laboratorio Turing;
    altrimenti
      1.1.2 studio;
2 finché(sono tra le 13 e le 14)
  2.1 lezione in aula B

```

Per esercitarci a vedere, passo dopo passo, quanto *complessa* e *rigidamente strutturata* possa diventare la descrizione, *annidiamo* le attività di martedì e di giovedì nella **seconda versione**, ottenendo la **terza versione**:

```

finché(la settimana è nel semestre)
1 se(è lunedì)
  1.1 attività lunedì;
2 se(è martedì)
  2.1 attività martedì:
    2.1.1 finché(sono tra le 9 e le 11)
      2.1.1.1 se(c'è esercitazione)
        2.1.1.1.1 lezione in aula B;
        altrimenti
        2.1.1.1.2 studio;
    2.1.2 finché(sono tra le 11 e le 13)
      2.1.1.1 lezione in aula B
    2.1.3 finché(sono tra le 14 e le 17)
      2.1.3.1 se(sono del turno T1)
        2.1.3.1.1 lezione in laboratorio Turing;
        altrimenti
        2.1.3.1.2 studio;
3 se(è mercoledì)
  3.1 attività mercoledì;
4 se(è giovedì)
  4.1 attività giovedì:
    4.1.1 finché(sono tra le 9 e le 12)
      4.1.1.1 se(sono del turno T2)
        4.1.1.1.1 lezione in laboratorio Turing;
        altrimenti
        4.1.1.1.2 studio;
    4.1.2 finché(sono tra le 13 e le 14)
      4.1.2.1 lezione in aula B
5 se(è venerdì)
  5.1 attività venerdì;
6 se(è sabato)
  6.1 attività sabato;
7 se(è domenica)
  7.1 attività domenica.

```

Esercizio 2 (Variante della terza versione) Proporre almeno un'alternativa alla **terza versione** appena definita che sia in grado di il medesimo insieme di attività, ma che sia strutturata come segue:

```

finché(settimana è nel semestre)
1 se(è lunedì):
  1.1 attività lunedì;
2 se(è martedì o giovedì):
  2.1 attività martedì-giovedì;
3 se( è mercoledì)
  3.1 attività mercoledì;
4 se(è venerdì)
  4.1 attività venerdì;
5 se(è sabato)
  5.1 attività sabato;
6 se(è domenica)
  6.1 attività domenica.

```

nella quale le alternative,inizialmente presenti, tra martedì e giovedì ora compaiono in una singola alternativa. Precisamente, si tratta di sviluppare la descrizione strutturata del punto “2.1 attività martedì-giovedì”.

Esercizio 3 (Completamento della terza versione) Completare la **terza versione**, o la sua variante, risultante dall'Esercizio 2 dell'attività settimanale di uno studente, componendo opportunamente sequenze di azioni, eventualmente iterate ed opportunamente selezionate a seconda delle necessità. ■

Domanda fondamentale. *Cosa impariamo dagli Esempi 1 e 2 nei quali abbiamo rigidamente strutturato possibili descrizioni di telefonate o attività settimanali?*

Risposte alla domanda fondamentale. Ci siamo avvicinati al modo in cui descriveremo algoritmi interpretabili meccanicamente da un interprete molto rigido come un calcolatore.

La strategia seguita per definire un algoritmo consiste in alcuni punti metodologici:

1. assumere di conoscere l'insieme di azioni atomiche a disposizione, ovvero di conoscere le azioni il cui significato sia dato per scontato, come, ad esempio “inserire cifra”;
2. definire azioni via via più complesse, mettendo in *sequenza*, oppure *iterando*, oppure *selezionando* azioni più semplici, già note.

Il processo di definizione di azioni complesse, componendo azioni più semplici già note tramite *sequenza*, *iterazione* o *selezione*, è un'applicazione dell'**approccio ricorsivo** alle soluzioni di problemi.

La struttura linguistica ricorsiva che adotteremo per descrivere *algoritmi* è catturata dalle seguenti clausole definitorie, tipicamente usate per descrivere grammatiche, ovvero oggetti formali che generano linguaggi:

$$A ::= P \mid A; A \mid \text{finché } (C) A \mid \text{se } (C) A \quad (1.1)$$

$$P ::= \{\text{Azioni primitive per l'interprete individuato}\} \quad (1.2)$$

$$C ::= \{\text{Condizioni che l'interprete individuato sa comprendere}\}. \quad (1.3)$$

La clausola (1.1) descrive la sintassi cui deve aderire la descrizione di un algoritmo: essa afferma che un algoritmo A è o un'azione primitiva P , o una sequenza tra due algoritmi, o l'iterazione di un algoritmo finché una condizione C è verificabile, o la selezione tra due algoritmi in base ad una qualche condizione C . La clausola (1.2) è l'insieme di azioni primitive che ci diamo come disponibili. La clausola (1.3) è l'insieme di condizioni che assumiamo poter identificare per decidere quando interrompere una iterazione o tra quali (sotto)algoritmi scegliere.

Esempio 3 Nei casi esemplificati sinora, P contiene *azioni primitive* come “inserire cifra” e “studio” mentre C contiene la descrizione di condizioni come “è lunedì”, “è martedì o giovedì” e “esistono cifre da immettere”. ■

Esercizio 4 (Azioni primitive e condizioni) Scorrendo gli algoritmi proposti negli Esempi 1 e 2, individuare tutte le azioni primitive che appartengono all'insieme P e tutte le condizioni che appartengono all'insieme C . ■

A conferma del fatto che quanto fatto si avvicina al modo in cui descriveremo algoritmi per calcolatori, quella che segue è una possibile ulteriore ristrutturazione della descrizione di una telefonata, sviluppata nell'Esempio 1:

```

1 public class Telefonata {
2
3     public static void rispondere() {
4         // una qualche definizione
5     }
6
7     public static void inserireCifra() {
8         // una qualche definizione
9     }
10
11    public static void parlare() {
12        // una qualche definizione
13    }
14
15    public static void attendereSquillo() {
16        // una qualche definizione
17    }
18
19    public static void attendereRisposta() {
20        if (libero())
21            while(numeroSquilli() < 4)
22                if(risposta())
23                    parlare();
24                else
25                    attendereSquillo();
26    }
27
28    public static void comporreNumero() {
29        while(mancanoCifre())
30            inserireCifra();
31    }
32
33    public static void main() {
34        comporreNumero();
35        attendereRisposta();
36    }
37 }
```

Nel listato qui sopra, è stata usata la sintassi del linguaggio di programmazione Java. In generale, la sintassi di linguaggi di programmazione reali contiene molti aspetti legati alla necessità di poterli interpretare per mezzo di un calcolatore. Il fattore comune tra molti linguaggi di programmazione, però, è la possibilità di descrivere algoritmi usando *solo* le strutture linguistiche (1.1), (1.2) e (1.3), avendo fissato azioni primitive come l'assegnazione di valori a variabili e condizioni primitive esprimibili attraverso opportuni calcoli su predicati.

Esercizio 5 (Descrizioni algoritmiche di attività quotidiane) Descrivere algoritmamente le seguenti attività:

1. servire al banco di un bar;
2. effettuare lavori domestici come stirare, lavare i piatti a mano o con la lavatrice;
3. giocare ad un qualche gioco da tavolo come le carte, gli scacchi.

Per ogni punto, affrontare la descrizione in accordo con lo schema seguito nell'Esempio 1. Il primo passo è identificare macro azioni distintamente separate l'una dall'altra. Le macro azioni possono essere raffinabili in (macro)azioni da mettere in sequenza, da iterare o tra cui scegliere. Alla fine la descrizione dell'algoritmo dovrebbe essere generabile tramite le clausole (1.1), (1.2) e (1.3), avendo fissato gli insiemi P e C . \square

Capitolo 2

Programmazione strutturata iterativa di base

Nel capitolo precedente abbiamo sostenuto che sia possibile descrivere algoritmi interpretabili dai calcolatori a patto di uniformarsi a strutture linguistiche specifiche, quali la sequenza, l'iterazione e la selezione le quali operano su insiemi di azioni primitive e di condizioni verificabili con cui costruire descrizioni di azioni via via più complesse.

Questo capitolo ha il duplice scopo di individuare l'insieme di azioni primitive che un calcolatore può interpretare e il linguaggio con cui esprimere condizioni che siano verificabili, in modo, ad esempio, da interrompere l'iterazione di un certo insieme di azioni.

2.1 Assegnazione

L'*assegnazione* è una delle possibili azioni primitive con cui costruire algoritmi adatti ad essere interpretati meccanicamente da un calcolatore elettronico programmabile. Riferendoci alle clausole (1.1), (1.2) e (1.3) che definiscono un algoritmo, le assegnazioni appartengono alla categoria sintattica P .

Esempio 4 Una possibile sequenza di assegnazioni è:

```
xZero = 21;  
xUno = 33;  
...  
xEnne = 45;
```

(2.1)

La sequenza (2.1) contiene un'assegnazione per riga. Gli aspetti essenziali di ogni assegnazione sono:

- A sinistra del simbolo “=” compaiono *identificatori di variabili* o più semplicemente *variabili*. Ciascuno tra `xZero`, `xUno`, ... va pensato come nome di un contenitore;
- Il simbolo “=” è detto di *assegnazione* ed indica proprio che il valore alla sua destra è associato alla variabile alla sua sinistra.
- A destra del simbolo di assegnazione può comparire una qualsiasi espressione che, valutata, possa fornire un valore, mentre a sinistra può comparire solo il nome di una variabile;

Quindi, ad esempio, invece di 45, nell'ultima assegnazione, a destra di “=”, può comparire $40 + 5$ o $33 + 12$. ■

2.1.1 Assegnazione e configurazioni

Parlare di “interpretazione di un'assegnazione” equivale a specificare senza ambiguità il suo effetto. Ogni assegnazione agisce su una *configurazione* di cui diamo una rappresentazione generica qui di seguito:

$$(\begin{array}{cccc} x_0 & x_1 & \dots & x_n \\ m_0, & m_1, & \dots, & m_n \end{array}) ,$$

e nella quale ad ogni *posizione* è assegnato un *nome di variabile*. Quindi, una *variabile* è il nome di una posizione all'interno di una configurazione. Tale posizione è un contenitore per valori, generalmente numerici. Il nome “variabile” non è occasionale. Esso esprime il fatto che i valori che essa contiene possano mutare nel tempo. La mutazione del valore è *sempre* conseguenza di un'assegnazione.

Esempio 5 (Interpretazione dell'assegnazione) Assumiamo che:

$$(\quad \text{xZero} \quad \text{xUno} \quad \dots \quad \text{xEnne} \quad) \\ (\quad ?, \quad ?, \quad \dots, \quad ?, \quad)$$

sia una configurazione alle cui posizioni assegniamo i nomi elencati, ma di cui non importano i valori.

La seguente sequenza di assegnazioni illustra l'effetto dell'interpretare un'assegnazione dopo l'altra, in accordo con l'ordine di apparizione dall'alto verso il basso:

```
(?, ?, ..., ?) /* configurazione da inizializzare */
xZero = 21;
(21, ?, ..., ?) /* prima componente inizializzata */
xUno = 33;
(21, 33, ..., ?)
...
xEnne = 45;
(21, 33, ..., 45)
xZero = 25; (2.2)
(25, 33, ..., 45)
xUno = xUno + 1;
(25, 33 + 1, ..., 45) .
```

Vale la pena osservare l'effetto di (2.2) e (2.3) sulla configurazione che le precede. L'Assegnazione (2.2) rimpiazza il valore 21 col valore 25, sottolineando l'idea che `xUno` sia il nome di un contenitore adatto a contenere valori che possono essere rimpiazzati.

L'assegnazione (2.3) sottolinea che il simbolo “=” non è da interpretare in maniera standard, ovvero non è una eguaglianza. In particolare, va interpretata come segue:

1. Prima occorre interpretare l'espressione a destra del simbolo “=”. Nel nostro caso, la valutazione richiede di leggere il valore nella variabile `xUno` che risulta essere 33;
2. Al valore risultante dalla lettura di `xUno`, al passo precedente, occorre sommare il valore 1, ottenendo 34;
3. Il valore ottenuto al passo precedente va rimpiazzato a quello esistente in `xUno`. Ovvero, 34 è rimpiazzato a 33. ■

2.1.2 Assegnazione e sequenza

Chiamiamo EXC2P il seguente problema:

Data una coppia (m, n) di valori interi produrre la coppia nella quale i valori sono scambiati. Ovvero, il risultato deve essere (n, m) .

Chiamiamo `a` la prima posizione della configurazione e `b` la seconda:

$$(\quad \text{a} \quad \text{b} \quad) \\ (\quad m \quad , \quad n \quad)$$

Per come abbiamo definito il comportamento dell'assegnazione, non è possibile applicare direttamente né:

$$\text{a} = \text{b};$$

alla configurazione iniziale, né:

$$\text{b} = \text{a}; .$$

Nel primo caso perderemmo irrimediabilmente il valore m in `a` e nel secondo il valore n in `b`. Per risolvere il problema occorre usare un'ulteriore elemento nella configurazione:

$$(\quad \text{a} \quad \text{b} \quad \text{tmp} \quad) \\ (\quad m \quad , \quad n \quad , \quad ? \quad)$$

il cui valore iniziale è ininfluente. Il problema EXC2P è, quindi, risolvibile grazie ad almeno la seguente sequenza di assegnazioni:

```
// (m, n, ?) in cui compaiono anche i valori iniziali da scambiare
tmp = a;
// (m, n, m)
a = b;
// (n, n, m)
b = tmp;
// (n, m, m) in cui la posizione più a destra è ormai inutile .
```

(2.4)

Esercizio 6 Scrivere l'altra possibile sequenza di assegnazioni che permetta di risolvere EXC2P. ■

2.2 Selezione e azioni condizionate

Chiamiamo MAX2P il problema che, data una coppia (m, n) di valori interi indichi quale sia il massimo tra m ed n .

Supponiamo di voler usare configurazioni di tre posizioni:

$$(\begin{array}{ccc} a & b & \text{max} \\ m & n & ? \end{array})$$

in cui le prime due componenti contengono i valori m ed n da confrontare e l'ultima il massimo tra i due. Una possibile soluzione a MAX2P è data da:

Algoritmo 1 per MAX2P

```
1: // (m, n, ?)
2: if (a > b) then
3:   // (m, n, ?)
4:   max = a;
5:   // (m, n, m)
6: else
7:   // (m, n, ?)
8:   max = b;
9:   // (m, n, n)
10: end if
```

L'Algoritmo 1 usa la selezione tra diverse azioni, usando la sintassi `if-then-else` (se-allora-altrimenti) dei linguaggi reali di programmazione. L'Algoritmo 1 si legge come segue:

- il comando di selezione, che comincia con la *parola riservata* “**if**”, confronta i valori associati ad **a** e **b**, per mezzo della condizione $a > b$:
 - se il risultato di $a > b$ è `true` — il valore in **a** è strettamente maggiore di quello in **b** —, allora si interpreta l'assegnazione `max = a;`. Si dice che “si segue il ramo **then** della selezione”;
 - altrimenti, se il risultato di $a > b$ è `false` — il valore in **a** è minore o uguale a quello in **b** —, allora si interpreta l'assegnazione `max = b;`. Si dice che “si segue il ramo **else** della selezione”;
- all'ingresso di ciascun ramo, la configurazione non è ancora cambiata perché non sono state interpretate assegnazioni;
- al termine di ciascun ramo la configurazione è cambiata in accordo con l'idea che **max** sia il nome della terza posizione. In essa c'è il valore di **a** se abbiamo seguito il ramo **then**, oppure il valore di **b** se abbiamo seguito il ramo **else**;
- `Massimo.java` è un programma Java che raffina l'Algoritmo 1, distinguendo i tre casi possibili del confronto tra i contenuti di **a** e **b** e stampando su standard output il massimo.

L'Algoritmo 2 è un ulteriore algoritmo per MAX2P. Esso sfrutta il fatto che è ammesso definire selezioni `if-then` (`se-allora`) in cui si esprime il solo ramo **then**, ovvero quello interpretato solo se l'espressione argomento della parola chiave **if** è vera.

Algoritmo 2 per MAX2P con selezioni ad un solo ramo

```

1: // (m, n, ?)
2: if (a > b) then
3:   // (m, n, ?)
4:   max = a;
5:   // (m, n, m)
6: end if
7: if (a <= b) then
8:   // (m, n, ?)
9:   max = b;
10:  // (m, n, n)
11: end if

```

Esercizio 7 Riformulare gli Algoritmi 1 e 2 in modo che le configurazioni usate abbiano solo due componenti e, conseguentemente, ipotizzando che il valore massimo, al termine degli algoritmi, sia nella posizione più a destra della configurazione, perdendo il valore n iniziale.

Esercizio 8 (Scambi di valori controllati da selezioni) • Chiamiamo MAX4P la seguente generalizzazione di MAX2P. Siano dati quattro numeri naturali m, n, o, p . L'obiettivo è conoscere il massimo tra i quattro.

Un algoritmo che risolva il problema, potrà assumere di manipolare configurazioni con almeno quattro posizioni a, b, c e d che conterranno i quattro valori dati.

L'esercizio consiste nello scrivere almeno un algoritmo che risolva MAX4P. Esistono vari approcci. Di seguito ne suggeriamo sommariamente un paio:

- La prima strategia di soluzione è così descritta: (i) la prima azione confronta il contenuto di a con quello di b . Nel caso il primo superi il secondo si scambiano i contenuti di a e di b ; (ii) la seconda azione confronta il contenuto di b con quello di c . Nel caso il primo superi il secondo si scambiano i contenuti di b e di c ; (iii) la terza azione confronta il contenuto di c con quello di d . Nel caso il primo superi il secondo si scambiano i contenuti di c e di d .

([EmersioneDelMassimoQuattroVars.java](#) è una classe Java che si comporta in accordo con la strategia appena descritta.)

- La seconda strategia di soluzione assume l'uso di configurazioni con 5 elementi dei quali l'ultimo è max. Essa è così descritta: (i) la prima azione assegna il contenuto di a a max. (ii) La seconda azione confronta il contenuto di b con quello di max. Se il primo supera il secondo allora il contenuto di b è copiato in max. (iii) La terza azione confronta il contenuto di c con quello di max. Se il primo supera il secondo allora il contenuto di c è copiato in max. (iv) La quarta azione confronta il contenuto di d con quello di max. Se il primo supera il secondo allora il contenuto di d è copiato in max.

- Siano dati 4 numeri naturali qualsiasi, memorizzati in una configurazione di quattro posizioni a, b, c, d . BS4P è il problema che, presa una quadrupla data, restituisce una quadrupla in cui i valori sono stati ordinati in ordine non decrescente leggendo a, b, c, d da sinistra verso destra.

([BubbleQuattroVars.java](#) è una classe Java che risolve BS4P.) ■

2.3 Ripetizione di azioni e costrutti iterativi

Chiamiamo SIDP il problema che, dati due numeri naturali $m, n \in \mathbb{N}$ ne restituisca la somma, sottostando al vincolo che le uniche operazioni disponibili per modificare un valore siano l'incremento od il decremento di una quantità data. Il vincolo imposto è da intendersi come segue. Ad esempio, sia dato il valore 3, e supponiamo di voler ottenere da esso il valore 5 comandando 2. Il valore finale 5 può essere ottenuto solo procedendo attraverso due incrementi successivi che, con il linguaggio aritmetico, scriverebbero $(3 + 1) + 1$. Analogamente, assumiamo che il valore 0, possa essere ottenuto da 3 solo attraverso tre decrementi successivi di una unità come indica l'espressione aritmetica $((3 - 1) - 1) - 1$.

In generale, questo significa che, per ogni $m, n \in \mathbb{N}$, i valori $m + n$ e $m - n$ potranno essere calcolati *solo* seguendo la struttura delle espressioni $(\dots(m + 1) + \dots 1)$ e $(\dots(m - 1) - \dots 1)$.

Per formulare una soluzione a SIDP è opportuno usare l'*iterazione* come, ad esempio, nell'Algoritmo seguente:

Algoritmo 3 per SIDP

```

1: a = 2;
2: b = 3;
3: while (a > 0) do
4:   a = a - 1;
5:   b = b + 1;
6: end while

```

Nell'Algoritmo 3 l'iterazione è descritta attraverso il costrutto `while-do`, tipico di linguaggi di programmazione reali e che rimpiazza il `finché ...` usato nella Sezione 1.2. L'Algoritmo 3 si legge come segue:

- Esso agisce su configurazioni di due elementi di nome `a` e `b` cui vengono assegnati i valori 2 e 3, rispettivamente, dalle istruzioni 1 e 2;
- Il costrutto iterativo comincia con la *parola riservata* `while`, seguita da un'espressione che, in questo specifico caso, confronta il valore associato ad `a`, col valore 0, per mezzo della condizione `a > 0`:
 - Se il risultato di `a > 0` è `true` — il valore in `a` è strettamente maggiore di 0 —, allora:
 1. nell'ordine in cui compaiono, si interpretano *tutte* le istruzioni che seguono `while` e che precedono `end while`. Nella nostra specifica situazione, siccome tra `while` e `end while` esistono esattamente due assegnazioni, interpretiamo prima `a = a - 1`; quindi `b = b + 1`. Si dice che “si esegue, o si interpreta, il corpo del `while`” e ...
 2. ... si riprende il flusso interpretando l'espressione `a > 0`, argomento della parola chiave `while`.
 - Altrimenti, se il risultato di `a > 0` è `false` — il valore in `a` è minore o uguale a 0 —, allora si ricomincia l'intepretazione dalla prima riga che segue `end while`. Si dice che “si esce dal corpo del `while`”.
- È, quindi, possibile che le istruzioni nel corpo del `while` vengano ripetutamente interpretate prima che l'espressione argomento della parola chiave `while` diventi `false`.

Nel caso in questione, l'evolversi delle configurazioni è rappresentato dalla Tabella 2.1.

N.ro di cicli percorsi	Conf. prima di 4	Conf. dopo 4 e prima di 5	Conf. dopo 5
0	(2, 3)	(1, 3)	(1, 4)
1	(1, 4)	(0, 4)	(0, 5)

Tabella 2.1: Evolversi delle configurazioni nell'Algoritmo 3, partendo col valore 2 in `a` e 3 in `b`.

Il valore 0 nella colonna “N.ro di cicli già percorsi” indica che non tutte le istruzioni che costituiscono il corpo `while` sono già state interpretate. Il valore 1 nella colonna “N.ro di cicli già percorsi” indica che la sequenza di tutte le istruzioni nel corpo del `while` è stata interpretata esattamente 1 volta, non essendo mai ripartiti dalla parola chiave `while`.

Se assegnassimo ad `a` un valore più grande di 2, potremmo anche scrivere il valore 2 nella colonna “N.ro di cicli già percorsi”, indicando che la sequenza di tutte le istruzioni nel corpo del `while` è stata interpretata esattamente 2 volte, dopo essere ripartiti dalla parola chiave `while` al termine del primo percorimento. E così via.

- La Tabella 2.1 non contiene ulteriori righe perché, una volta raggiunta la configurazione (0, 5) significa che `a` vale 0. Quindi, `a > 0` vale `false` e si deve proseguire dalla prima istruzione che segue `end while`. Siccome nessuna istruzione esiste al di sotto di `end while` la configurazione consegnataci dall'Algoritmo 3 è (0, 5) in cui la prima componente, individuata da `a`, contiene il risultato cercato.
- [SommaSuccessoreIterato.java](#) è un programma Java che implementa l'Algoritmo 3 per SIDP.

Esercizio 9 (Problema PPID) Fornire un algoritmo che risolva il problema PPID, ovvero che calcoli il valore di `a` elevato a `b`, supponendo però di saper solo calcolare il prodotto tra due numeri ed il predecessore di un numero. ■

Quoziente e resto per iterazioni successive di sottrazioni

Progettiamo daccapo un algoritmo per risolvere il seguente problema QRSP:

Siano dati i valori n e d , rispettivamente numeratore e denominatore. L'obiettivo è fornire un algoritmo che restituisca il quoziente q ed il resto r della divisione intera tra n e d .

Ad esempio, se $n = 7$ e $d = 2$, allora la divisione intera $7/2$ deve restituire quoziante $q = 3$ e resto $r = 1$. Un algoritmo utilizzabile è quello che, alle scuole elementari, viene utilizzato per illustrare il meccanismo della divisione. Ovvero la divisione è una distribuzione di elementi tra destinatari possibili: 7 “steccoline” vanno distribuite equamente a 2 destinatari, senza creare differenze, ovvero, non assegnando steccoline rimanenti se non ne esistono abbastanza per essere distribuite a tutti.

Segue una rappresentazione schematica del processo di distribuzione:

Steccoline							Steccoline							Steccoline							Steccoline						
1 2 3 4 5 6 7							3 4 5 6 7							5 6 7							7						
Dest. 1		Dest. 2		Dest. 1		Dest. 2		Dest. 1		Dest. 2		Dest. 1		Dest. 2		Dest. 1		Dest. 2		Dest. 1		Dest. 2					

Da sinistra, inizialmente, nulla è stato distribuito. Quindi, si distribuiscono due steccoline, una per destinazione. Si ripete l'operazione finché non rimane una singola steccolina che non è più equamente distribuibile. Il numero conclusivo di distribuzioni effettuate rappresenta il quoziante. La singola steccolina *indistribuibile* è il resto.

La configurazione di partenza ragionevole contiene quanto necessario a raccogliere passo dopo passo sia il quoziante q , sia il resto r , dati il numeratore n ed il denominatore d . La configurazione iniziale può, quindi, avere forma:

$$(\begin{array}{cccc} n & d & q & r \\ n , & d , & 0 , & n) \end{array}) .$$

In particolare, se dobbiamo dividere 7 (steccoline) per 2 (destinatari) la configurazione iniziale è:

$$(\begin{array}{cccc} n & d & q & r \\ 7 , & 2 , & 0 , & 7) \end{array}) .$$

L'Algoritmo (4) costruisce via via configurazioni delle quali l'ultima conterrà il quoziante e resto della divisione del valore inizialmente nel numeratore per il valore inizialmente nel denominatore:

Algoritmo 4 per QRSP

```

1: q = 0;
2: r = 7;
3: // (7, 2, 0, 7)
4: while (r >= d) do
5:   q = q + 1;
6:   r = r - d;
7: end while

```

Nello specifico, supponiamo n sia il nome della variabile col valore 7 del numeratore e d il nome della variabile col valore 2 del denominatore.

L'Algoritmo (4) si legge come segue:

- L'assunzione sui valori del numeratore e del denominatore, più le assegnazioni $q = 0$; e $r = 7$; creano la configurazione iniziale che leggiamo alla linea 3.
- L'espressione di terminazione $r >= d$ del costrutto iterativo **while**, implica che il corpo dell'iterazione venga eseguito solo quando il valore in r non sia inferiore a quello in d . Quindi:
 - Se il risultato di $r >= d$ è **true** — il valore in r non è strettamente inferiore a d —, allora:
 1. nell'ordine in cui compaiano, interpretiamo tutte e sole le assegnazioni $q = q + 1$; quindi $r = r - d$; nel corpo del **while** e ...
 2. ... riprendiamo il flusso dell'interpretazione dall'espressione $r >= d$, argomento di **while**.
 - Altrimenti, se il risultato di $r >= d$ è **false** — il valore in r è minore di quello in d —, allora si esce dal corpo del **while**".
- È, quindi, possibile che le istruzioni nel corpo del **while** vengano ripetutamente interpretate prima che l'espressione argomento della parola chiave **while** diventi **false**.

Nel caso in questione, l'evolversi delle configurazioni è rappresentato dalla Tabella 2.2.

N.ro di cicli già percorsi	Conf. prima di 5	Conf. dopo 5 e prima di 6	Conf. dopo 6
0	(7, 2, 0, 7)	(7, 2, 1, 7)	(7, 2, 1, 5)
1	(7, 2, 1, 5)	(7, 2, 2, 5)	(7, 2, 2, 3)
2	(7, 2, 2, 3)	(7, 2, 3, 3)	(7, 2, 3, 1)

Tabella 2.2: Evolversi delle configurazioni nell’Algoritmo 4, partendo col valore 7 in a e 2 in b.

Il valore 0 nella colonna “N.ro di cicli già percorsi” indica che non tutte le istruzioni che costituiscono il corpo **while** sono già state interpretate. Il valore 1 nella colonna “N.ro di cicli già percorsi” indica che la sequenza di tutte le istruzioni nel corpo del **while** è stata interpretata esattamente 1 volta, non essendo mai ripartiti dalla parola chiave **while**. Il valore 2 nella colonna “N.ro di cicli già percorsi” indica che la sequenza di tutte le istruzioni nel corpo del **while** è stata interpretata esattamente 2 volte, dopo essere ripartiti dalla parola chiave **while** al termine del primo percorso. Il valore 3 nella colonna “N.ro di cicli già percorsi” indica che la sequenza di tutte le istruzioni nel corpo del **while** è stata interpretata esattamente 3 volte, essendo ripartiti dalla parola chiave **while** al termine sia del primo, sia del secondo percorso. E così via.

- La Tabella 2.2 non contiene ulteriori righe perché, una volta raggiunta la configurazione (7, 2, 3, 1) significa che il valore in r è inferiore a quello in d . Quindi, $r \geq d$ vale **false** e si deve proseguire dalla prima istruzione che segue **end while**. Siccome nessuna istruzione esiste al di sotto di **end while** la configurazione consegnataci dall’Algoritmo 4 è (7, 2, 3, 1) in cui la terza componente, individuata da q , contiene il quoziente e la quarta, ovvero r , contiene in resto della divisione intera del valore in q per il valore in n .

Esercizio 10 Scrivere algoritmi che risolvano i seguenti problemi:

1. PDSP: “Dato un numero intero m , determinare se esso sia pari o dispari, iterando solo sottrazioni.”
2. PDMP: “Determinare se il valore assegnato ad una variabile sia pari o dispari, sfruttando l’operatore modulo ‘%’ che si comporta come segue: $n \% 2$ vale 0 se n è divisibile per 2 senza resto, altrimenti vale 1.”
3. SPNP: “Fissato un naturale n , calcolare la somma dei primi numeri naturali, senza sfruttare la nota formula $\frac{n*(n+1)}{2}$.”
4. MSMP: “Fissati due numeri naturali m ed n , determinare il minore, supponendo d’avere a disposizione solo le seguenti operazioni di base: decremento di una unità, risposta positiva o negativa alla domande $n = 0?$, $m = 0?$, $n \neq 0?$, $m \neq 0?$.” ■

2.3.1 Quadrato di un numero intero basato sui prodotti notevoli

Definiamo il problema QPNP (Quadrato attraverso prodotti notevoli).

Vogliamo calcolare il quadrato di un numero intero n , supponendo alcune ipotesi restrittive: (i) non sappiamo calcolare la moltiplicazione tra numeri naturali arbitrari, ma (ii) sappiamo raddoppiare numeri e (iii) sappiamo incrementarli di una unità o decrementarli di quantità arbitrarie. Sappiamo, inoltre che se $n = 0$, allora n^0 vale 1.

I vincoli imposti sull’uso delle operazioni significa, ad esempio che non possiamo calcolare n^2 semplicemente usando $n * n$. Questo forza l’uso di una qualche strada alternativa. Una possibile è la seguente:

$$(n + 1)^2 = (n + 1) \cdot (n + 1) = n^2 + 2 \cdot n + 1 . \quad (2.5)$$

Possiamo convincerci — non dimostrare! (almeno per ora) — della validità di (2.5) usando qualche esempio. Supponiamo $n + 1 = 3$. Allora:

$$\begin{aligned} 3^2 &= (2 + 1)^2 \\ &= 2^2 + 2 \cdot 2 + 1 . \end{aligned} \quad (2.6)$$

Ma 2^2 coincide con $(1 + 1)^2$, quindi possiamo riapplicare (2.5):

$$\begin{aligned} (1 + 1)^2 &= 1^2 + 2 \cdot 1 + 1 \\ &= 4 . \end{aligned} \quad (2.7)$$

Quindi (2.6) produce effettivamente il risultato atteso, col processo di calcolo seguente:

$$\begin{aligned} 3^2 &= (2 + 1)^2 \\ &= 2^2 + 2 \cdot 2 + 1 \quad (\text{usando (2.7)}) \\ &= 4 + 4 + 1 \\ &= 9 . \end{aligned}$$

Se generalizziamo il calcolo sviluppato dall'esempio, otteniamo quanto segue:

$$\begin{aligned}
 (n+1)^2 &= n^2 + 2 \cdot n + 1 \\
 &= ((n-1)+1)^2 + 2 \cdot n + 1 \\
 &= ((n-1)^2 + 2 \cdot (n-1) + 1) + 2 \cdot n + 1 \\
 &= (((n-2)+1)^2 + 2 \cdot (n-1) + 1) + 2 \cdot n + 1 \\
 &= (((n-2)^2 + 2 \cdot (n-2) + 1) + 2 \cdot (n-1) + 1) + 2 \cdot n + 1 \\
 &= \dots \\
 &= (n-n)^2 + (2 \cdot (n-n) + 1) + \dots + (2 \cdot (n-1) + 1) + (2 \cdot n + 1) \\
 &= 0 + \sum_{i=0}^n (2 \cdot (n-i) + 1) \\
 &= \sum_{i=0}^n (2 \cdot (n-i)) + \sum_{i=0}^n 1 \\
 &= (n+1) + 2 \cdot \sum_{i=0}^n (n-i) \\
 &= (n+1) + 2 \cdot \sum_{i=0}^{n-1} (n-i) . \tag{2.8}
 \end{aligned}$$

La sommatoria $\sum_{i=0}^{n-1} (n-i)$ esprime la necessità di accumulare la somma dei valori $n, n-1, n-2, \dots, 1$ in una opportuna variabile. Nell'ultimo punto dell'Esercizio 10 abbiamo già visto come si accumuli un valore in una variabile, che possiamo chiamare `s`. Una volta che $n + (n-1) + (n-2) + \dots + 1$ si trovi in `s`, seguendo (2.8), ne raddoppiamo il valore e vi sommiamo il contenuto di `n`, ottenendo n^2 in `s`.

L'Algoritmo 5 implementa quanto descritto, assumendo che la variabile `n` contenga un valore naturale n . La linea

Algoritmo 5 per QPNP

```

1: s = 0;
2: i = 1;
3: // (n, 1, 0)
4: while (i < n) do
5:   s = s + (n - i);
6:   i = i + 1;
7: end while
8: // (n, n, (n-1)+(n-2)+...+(n-(n-2))+(n-(n-1)))
9: // ovvero (n, n, ∑_{i=0}^{n-1} (n-i))
10: s = n + 2 * s;
11: // (n, n, n+2 · ∑_{i=0}^{n-1} (n-i))

```

3, illustra la configurazione iniziale in cui la prima componente corrisponde alla variabile `n`, col valore n di cui calcolare il quadrato, la seconda ad `i` e la terza ad `s`.

La Tabella 2.3 illustra come si evolvono le configurazioni dell'Algoritmo 5.

N.ro di cicli già percorsi	Conf. prima di 5	Conf. dopo 5 e prima di 6	Conf. dopo 6
0	(3, 1, 0)	(3, 1, 2)	(3, 2, 2)
1	(3, 2, 2)	(3, 2, 3)	(3, 3, 3)

Tabella 2.3: Evolversi delle configurazioni nell'Algoritmo 5, partendo col valore 3 in `n`, ovvero con $n = 3$.

Il valore 0 nella colonna "N.ro di cicli già percorsi" indica che non tutte le istruzioni che costituiscono il corpo `while` sono già state interpretate. Il valore 1 nella colonna "N.ro di cicli già percorsi" indica che la sequenza di tutte le istruzioni nel corpo del `while` è stata interpretata esattamente 1 volta, non essendo mai ripartiti dalla parola chiave `while`.

Nel caso in esame, la seconda componente nella configurazione contiene valore 3 che rende falsa la condizione $3 < 3$. La terza contiene il valore 3 della sommatoria. L'assegnazione alla linea 10 modifica 3, , raddoppiandolo e sommandovi 3. La configurazione finale è $(3, 3, 9)$, la cui ultima componente contiene 9, ovvero il valore cercato.

Nota 1

La potenziale rilevanza della strategia di soluzione per QPNP sta nel fatto che utilizza operazioni le quali, viste al livello della CPU, sono poco costose da eseguire. Infatti, nella rappresentazione in base 2 dei numeri, la moltiplicazione per 2 equivale ad uno spostamento verso sinistra di tutti i bit, operazione molto veloce. Sottrazioni di valori crescenti di una singola unità da uno stesso valore, sono ottimizzabili “semplicemente”. Ma queste sono proprio le operazioni che abbiamo individuato come necessarie per il calcolo di n^2 . ■

Esercizio 11 1. MIDP è un problema analogo a SIDP. In esso l’obiettivo è ottenere la differenza tra due interi $m, n \in \mathbb{N}$, supponendo di saper eseguire solo il decremento di una singola unità ad ogni numero naturale.

Ad esempio, se n è 3, allora assumiamo di saper dire che 3-1 valga 2. Tuttavia, per dare significato diretto a 3-2 occorre “spezzare” il problema, riconducendolo a (3-1)-1. Rispetto a SIDP, esiste un ulteriore vincolo. La differenza tra m ed n deve essere effettuata solo se $m \geq n$.

2. SIDP' ha lo stesso enunciato di SIDP. Occorre individuare l’operazione che, partendo da una opportuna configurazione, contenente $m, n \in \mathbb{N}$, permetta di ottenerne una finale contenente $m + n$, sempre assumendo di saper solo eseguire incrementi e decrementi di una unità sugli elementi delle configurazioni. L’ulteriore vincolo di SIDP', rispetto a SIDP, è che le configurazioni debbano avere tre componenti, tutte rilevanti per l’ottenimento del risultato.
3. SIDP'' ha lo stesso enunciato di SIDP'. Il vincolo è nuovamente sulla forma delle configurazioni. Esse devono avere cinque componenti, due delle quali servano a non distruggere i valori iniziali m ed n di cui calcolare la somma. Fornire un paio di *istanze* del problema e di sequenze di azioni che producano il risultato voluto.
4. MSDP è relativo alla moltiplicazione tra numeri. Dati $m, n \in \mathbb{N}$, individuare la struttura delle configurazioni e le azioni necessarie a permettere il calcolo di $n * m$, assumendo di saper eseguire *somme generiche* tra numeri, e decrementi di una singola unità su ogni numero. Ad esempio, $m * 3$ può solo essere calcolata come $(m + m) + m$. Come in SIDP, ad esempio, $n - 2$ può solo essere il risultato di $(n - 1) - 1$.
5. MSDP' è analogo a MSDP, ma con una limitazione ulteriore. L’unico modo per sommare due numeri è applicare incrementi o decrementi di una singola unità ai valori coinvolti nelle configurazioni. Fornire un paio di *istanze* del problema e di sequenze di azioni che producano il risultato voluto. ■

2.3.2 Iterazioni di iterazioni

Vale la pena osservare che esistono problemi i quali possano richiedere l’*annidamento* di iterazioni, esattamente come altri possano richiedere l’annidamento di selezioni, come nelle soluzioni da dare ai problemi dell’Esercizio 8.

Un problema da risolvere annidando un paio di iterazioni è MSDP' dell’Esercizio 11.

L’Algoritmo 6 si comporta come richiesto: ovvero osa solo espressioni in cui si incrementi o decrementi di una unità

Algoritmo 6 per MSDP’

```

1:   r = 0;
2:   i = 0;
3: // (3, 2, 0, 0)
4: while (n > 0 && i = 0) do
5:   n = n - 1;
6:   i = m;
7:   while (n >= 0 && i > 0) do
8:     i = i - 1;
9:     r = r + 1;
10:  end while
11: end while
12: // (3, 0, 0, 6)
```

il valore di una variabile.

La Tabella 2.4 raccoglie le configurazioni via via costruite dall’Algoritmo 6 sotto l’ipotesi che la variabile m contenga il valore 3 e la variabile n contenga il valore 2.

È possibile rileggere il modo in cui la Tabella 2.4 sia costruita come segue:

- Il corpo del ciclo più esterno contiene una sequenza di due assegnazioni ed una iterazione. Eseguire il corpo del ciclo più esterno significa interpretare le due assegnazioni ed interpretare tante volte quanto è necessario il corpo del ciclo interno.
- Il corpo del ciclo più interno contiene solo due assegnazioni. Esse sono interpretate un numero di volte che dipende dal valore in i, il quale è reinizializzato al valore di m ogni volta che si re-interpreta daccapo il corpo del ciclo più esterno.

N.ro di cicli esterni	Conf. prima di 5	Conf. dopo 6	N.ro di cicli interni	Conf. prima di 8	Conf. dopo 9
0	(3, 2, 0, 0)	(3, 1, 3, 0)	0	(3, 1, 3, 0)	(3, 1, 2, 1)
			1	(3, 1, 2, 1)	(3, 1, 1, 2)
			2	(3, 1, 1, 2)	(3, 1, 0, 3)
1	(3, 1, 0, 3)	(3, 0, 3, 3)	0	(3, 0, 3, 3)	(3, 0, 2, 4)
			1	(3, 0, 2, 4)	(3, 0, 1, 5)
			2	(3, 0, 1, 5)	(3, 0, 0, 6)

Tabella 2.4: Evolversi delle configurazioni nell’Algoritmo 6, partendo col valore 3 in m e 2 in n , ovvero assumendo $m = 3, n = 2$.

2.3.3 Riferimenti bibliografici

Vari capitoli de [SM14] coprono gli argomenti trattati da questa parte del programma didattico, non necessariamente nello stesso ordine sviluppato a lezione: [SM14, Capitolo 1], Sezioni 1.3.2, 1.3.3, 1.3.4, [SM14, Capitolo 2], Sezione 2.1, [SM14, Capitolo 3], Sezione 3.1, [SM14, Capitolo 4], Sezioni 4.1, 4.2.

Capitolo 3

Metodi e Modello di gestione della memoria

Illustreremo aspetti della gestione della memoria da parte della *java virtual machine*, che risponde al comando `java` e che usiamo per interpretare un qualsiasi *file* oggetto con estensione `class`, prodotto dall'applicazione del compilatore `javac` ad un *file* sorgente con estensione `java`.

Vedremo che le classi possono contenere più metodi tra i quali non necessariamente si trova `main`.

Parleremo di passaggio di parametri che abbiano un tipo di base, distinguendo tra quelli formali e quelli attuali.

Descriveremo la struttura *frame*, associata a ciascun metodo per mantenerne le variabili locali, e la struttura *frame stack* costituita da una sequenza di *frame*, gestita in accordo con una politica *last-in-first-out* (LIFO).

Gli argomenti sono trattati anche in [SM14, Capitolo 5], Sezioni 5.1, 5.2 e 5.3.

3.1 *Frame* e variabili locali al `main`

Supponiamo di aver compilato la classe seguente, per ipotesi memorizzata nel *file* `SoloMain.java`:

```
1 public class SoloMain {  
2  
3     public static void main (String[] args) {  
4  
5         int      a = 1;  
6         boolean b = (a == a);  
7  
8         System.out.println(b);  
9     }  
}
```

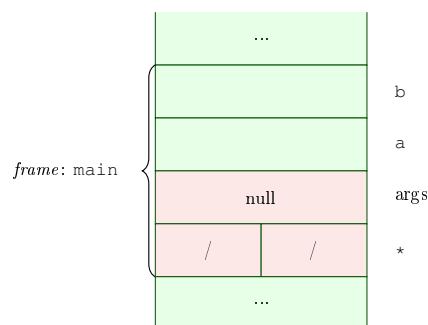
Questo significa avere a disposizione il file oggetto `SoloMain.class`.

Il comando `java SoloClass` guida il PC all'interpretazione del codice in `SoloMain.class` corrispondente al metodo `main`.

Allocazione. Prima di qualsiasi altra operazione avviene l'*allocazione* di un insieme di *Byte*.

L'insieme è detto *frame attivo*.

“Allocare *Byte*” significa “riservare *Byte*” per uno specifico scopo. Lo scopo è memorizzare i valori assunti dalle variabili che compaiono nel `main` durante l'interpretazione. Quindi, il *frame* allocato per il `main` ha la seguente struttura:



che analizziamo nel dettaglio, partendo dal basso:

- la cella di indirizzo * è la prima allocata. Essa è logicamente divisa in due parti, seguendo la notazione del testo [SM14]. Per ora, senza ulteriori dettagli, è sufficiente dire che:

- una parte permetterà di recuperare un valore calcolato,
 - l'altra parte conterrà, idealmente, un numero di linea del sorgente.
- Anche le altre celle hanno un indirizzo.

Al nostro livello di dettaglio, come indirizzo usiamo il nome della variabile che ciascuna cella rappresenta:

- a indica la cella che sarà usata per contenere i valori assunti da a durante l'interpretazione e
- b indica la cella che sarà usata per contenere i valori assunti da b.

Notiamo che né la cella a, né b contengano alcun valore. I valori saranno scritti solo come conseguenza dell'interpretazione delle righe che compongono il metodo main.

- Rimane la cella di indirizzo args. Essa rappresenta il *parametro formale* del metodo main che troviamo scritto tra parentesi, assieme al suo tipo String[] nella *signature*:

```
main (String[] args)
```

contenuta nella *dichiarazione* del metodo:

```
public static void main (String[] args)
```

Per ora anticipiamo solo il necessario per comprendere quel che sta succedendo:

- un parametro formale è una variabile il cui valore dipende dal “mondo esterno” al metodo. Vedremo in seguito quando e come il valore di un parametro formale venga fissato. L'importante è cominciare a ricordare che viene allocato lo spazio necessario a contenere i valori per ogni parametro formale di un dato metodo.
- String[] indica il tipo del parametro esattamente come, ad esempio, in int a = 1; dove il prefisso int specifica il tipo della variabile a.

Il punto è che String[] non è un tipo primitivo.

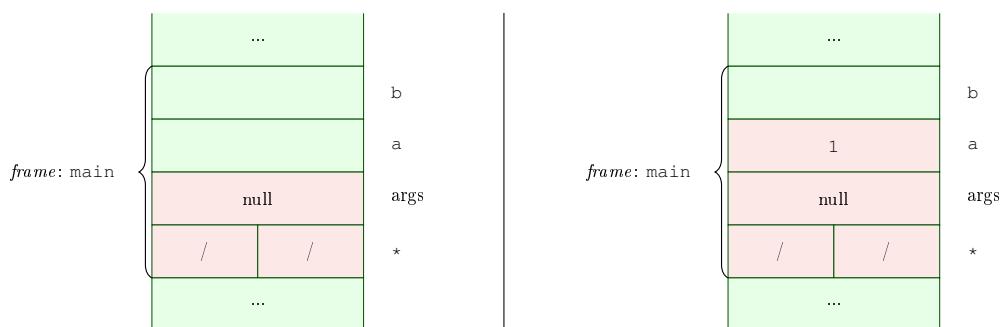
Per i tipi non primitivi, il meccanismo di allocazione dello spazio per contenere valori appropriati non è diretto come quello usato per l'allocazione di valori di tipi base. Vedremo in seguito i dettagli.

Per ora, intuitivamente, la parola chiave null esplicita che args non assume alcun valore (non primitivo) utilizzabile

Interpretazione. Solo ad allocazione avvenuta, comincia l'interpretazione vera e propria delle linee di codice in SoloMain.class corrispondenti a quelle in SoloMain.java. Quel che succede è esattamente quel che possiamo aspettarci, interpretando linea per linea il sorgente, come segue:

Linea 5. Occorre interpretare un'assegnazione. Valutiamo l'espressione a destra del simbolo =, ottenendo il valore 1. Esso va memorizzato nella cella di indirizzo a, omonimo alla variabile.

Partendo dalla configurazione a sinistra, terminiamo col *frame* sulla destra:



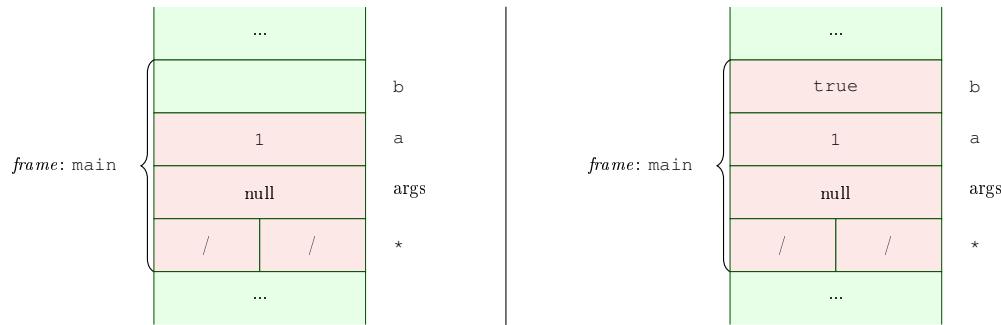
Linea 6. Occorre interpretare un'assegnazione. Valutiamo l'espressione a destra del simbolo =. La valutazione è meno immediata di quella precedente perché siamo di fronte ad una espressione “complessa” a == a.

Per valutarla dobbiamo risalire al valore della variabile a, cominciando a cercarne l'esistenza proprio dal *frame attivo*.

Nel caso in questione, il *frame attivo* è relativo al metodo main.

In esso troviamo a e da essa ricaviamo il valore 1. Ora che conosciamo il valore associato ad a possiamo valutare l'espressione 1 == 1. Siccome i valori confrontati sono identici, il risultato è true.

Partendo dalla configurazione a sinistra, terminiamo col *frame* sulla destra:



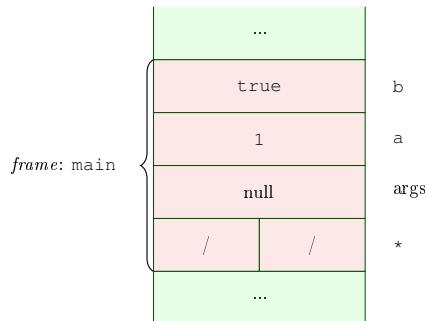
Linea 8. Occorre interpretare il richiamo al metodo il cui nome è `System.out.println` cui passiamo come argomento il valore contenuto in una variabile di nome `b`.

Come per la valutazione dell'espressione `a == a`, cerchiamo l'esistenza di una qualche variabile di nome `b` nel *frame attivo*. Siccome `b` esiste e contiene `true`, possiamo interpretare:

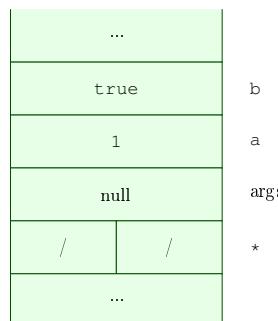
```
System.out.println(true);
```

ottenendo la pubblicazione della costante `true` sullo standard output, ovvero, tendenzialmente, sullo schermo del PC in uso.

La situazione del *frame attivo* non cambia, rimanendo:



Disallocazione. Al di sotto della riga 8 non rimangono istruzioni significative da interpretare. Il compito della *java virtual machine* è *disaloccare* lo spazio riservato al *frame attivo*, ripristinando lo stato quasi identico a quello precedente all'allocazione. Si ritorna, infatti, alla situazione:



Le celle contengono ancora i valori risultanti dall'interpretazione del metodo `main`. Tuttavia, esse sono disponibili per memorizzare valori, non appena siano nuovamente coinvolte nella formazione di un *frame*.

3.2 Metodi senza risultato, ma con parametri

Interpretiamo ora il sorgente `MetodoArgomento.java`:

```

1 public class MetodoArgomento {
2     public static void m (int b) {
3         int a = b + 4;
4     }
5
6     public static void main (String[] args) {
7         int a = 1;
8         int b = 2;
9     }
10 }
```

```

10     m(b + 3);
11     b = b + 1;
12 }

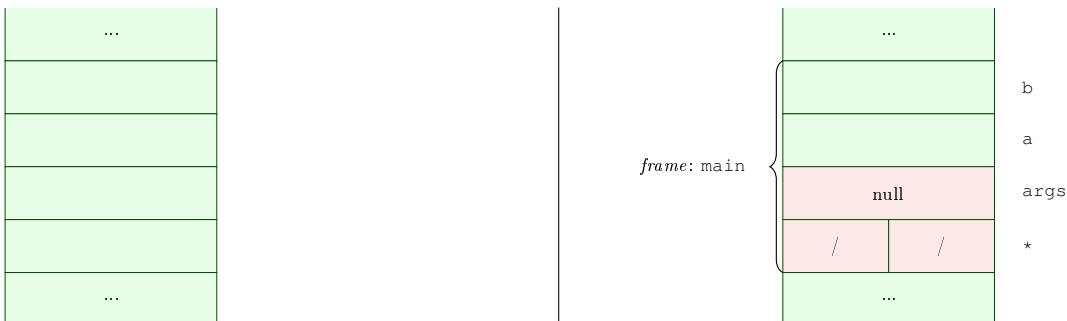
```

cui corrisponderà l'oggetto `MetodoArgomento.class` prodotto per mezzo di `javac`.

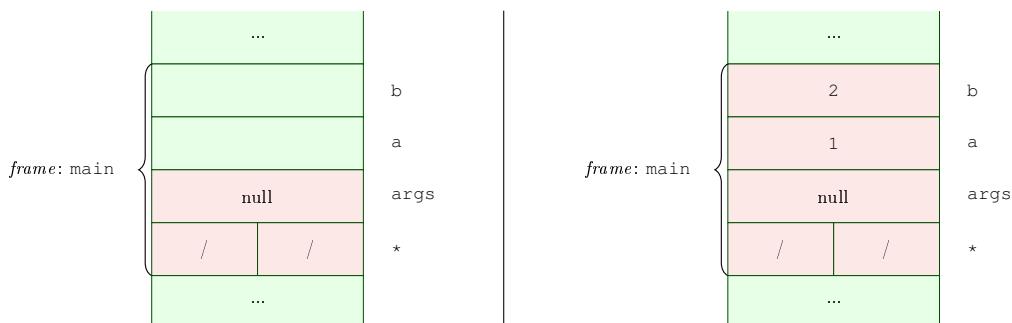
L'interpretazione comincia con l'allocazione del *frame* relativo al `main`, che diventa quello *attivo*, e prosegue con l'interpretazione delle sue linee di codice sino a quella contenente `m(b + 3);`. Il metodo `m` è interpretato in maniera del tutto analoga a `main`. Occorre allocare un *frame*, che diventa quello *attivo*, e che deve essere adatto a contenere i valori delle variabili necessarie ad `m`. Tra le variabili del *frame* per `m` c'è anche il *parametro formale* `b` che dovrà assumere il corretto valore. Una volta allocato il *frame* per `m` se ne interpretano tutte le istruzioni, in analogia a quanto faremmo con le linee di codice nel `main`. Al termine di `m` si disalloca il *frame* ad esso relativo. Quindi, il *frame attivo* ridiventa quello di `main`. Il *punto di rientro* è la stessa istruzione che ha generato l'allocazione del *frame* per `m`, ovvero la chiamata `m(b + 3)` al metodo `m`. Quindi, si procede, interpretando la parte di `main` che segue il richiamo del metodo `m`.

Discutiamo il dettaglio, illustrando l'evoluzione dell'utilizzo della memoria.

Allocazione main. Interpretando la riga 6 passiamo dalla configurazione a sinistra a quella destra:



Interpretazione main. L'interpretazione delle righe 7 ed 8 modificano il *frame*, passando dalla configurazione di sinistra a quella sulla destra:



Allocazione m. La riga 9 contiene, in gergo, una *chiamata al metodo m, usando come parametro attuale il risultato dell'espressione b + 3*.

Se la riga 9 fosse, ad esempio:

```
m(3);
```

essa conterebbe una *chiamata al metodo m, usando come parametro attuale il risultato dell'espressione 3*.

L'interpretazione corretta di una chiamata procede in accordo con i seguenti passi:

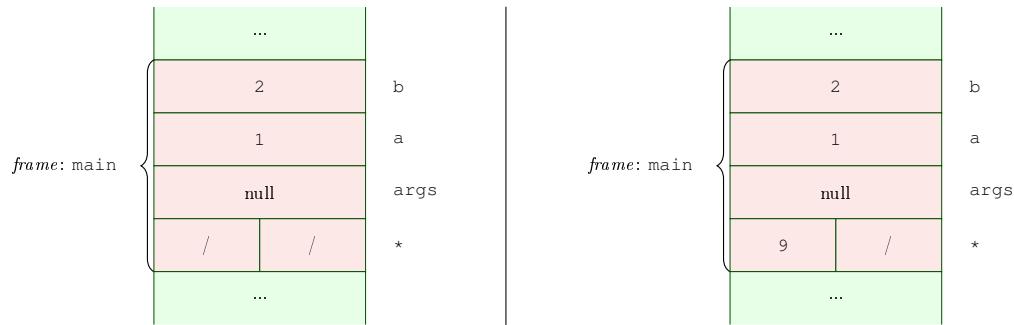
- la prima azione consiste nel valutare il parametro attuale della chiamata, ovvero l'espressione che si trova tra le parentesi che individuano l'argomento del metodo chiamato.

Nel nostro caso, la riga chiamante è `m(b + 3);` e l'espressione che costituisce il parametro attuale è `b + 3`.

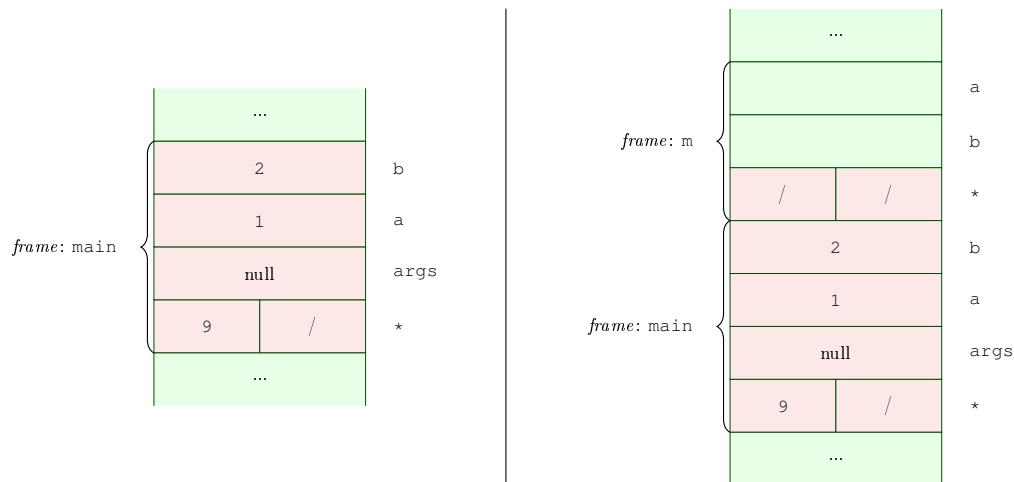
La sua valutazione richiede di individuare il valore della variabile `b`. La cerchiamo a partire dal *frame attivo* che è anche l'unico disponibile, per il momento.

Siccome, nel *frame attivo*, `b` vale 2, il valore del parametro attuale, ovvero dell'espressione `b + 3`, è 5.

- la seconda azione è ancora sul *frame attivo* del *metodo chiamante*, nel nostro caso `main`. Si memorizza la riga da cui occorrerà riprendere l'interpretazione del codice di `main` in una delle due posizioni della cella `*`. Passiamo dalla configurazione di sinistra a quella sulla destra:



- la terza azione è l'operazione naturale, ovvero l'allocazione del *frame* relativo al metodo chiamato che, nel nostro caso, è *m*. Il metodo *m* non potrebbe gestire correttamente le variabili di sua competenza senza *frame* dedicato e che diventi quello *attivo*. Passiamo dalla configurazione di sinistra a quella sulla destra:

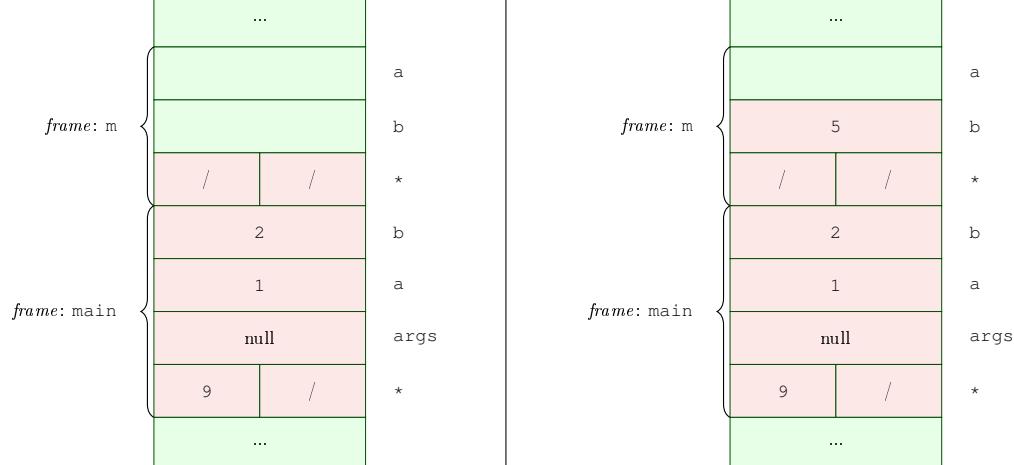


Osservazioni cruciali sono:

- il *frame* di *m* è idealmente “sopra” quello del *main*. Questo significa rispettare la politica di allocazione LIFO dei *frame*, già accennata. Ne vedremo tra poco il significato.
- Il *frame* di *m* può contenere variabili che sembrano avere lo stesso nome di quelle del *main*. In realtà le variabili non sono omonime perché, per individuarle, quel che conta è anche il *frame* nel quale sono definite.
- Esattamente come per il parametro formale *args* di *main*, è stato allocato dello spazio per il parametro formale *b* di *m*.
- Una volta allocato il *frame* per *m* si assegna al parametro formale il valore del parametro attuale.

Nel nostro caso il parametro formale è l'argomento tra parentesi *b* nella *signature* *m(int b)* di *m*. Inoltre, sappiamo già che il valore del parametro attuale, ovvero dell'espressione *b + 3* in *m(b + 3)*, è 5.

Assegniamo 5 a *b* nel *frame* di *m*, passando dalla configurazione di sinistra a quella sulla destra:



Ora che l'allocazione del *frame* di *m* è completa, possiamo interpretare *m*, guardando al suo *frame* come a quello *attivo*.

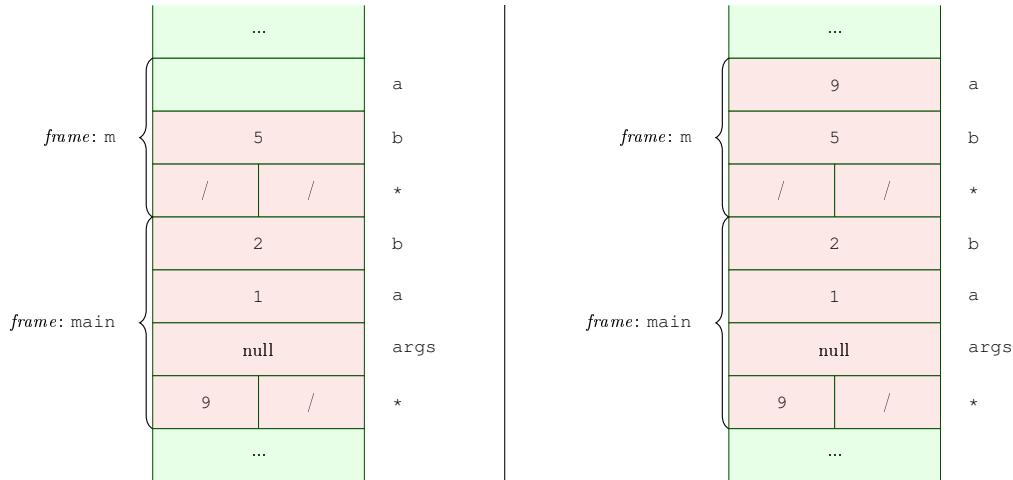
Interpretazione di m. Consiste nell'interpretare l'unica assegnazione a riga 3. Il primo passo è l'interpretazione dell'espressione a destra del simbolo uguale che dipende dal valore della variabile b.

La variabile b, che nel nostro caso è anche il parametro formale, va cercata tra quelle disponibili nel *frame attivo*.

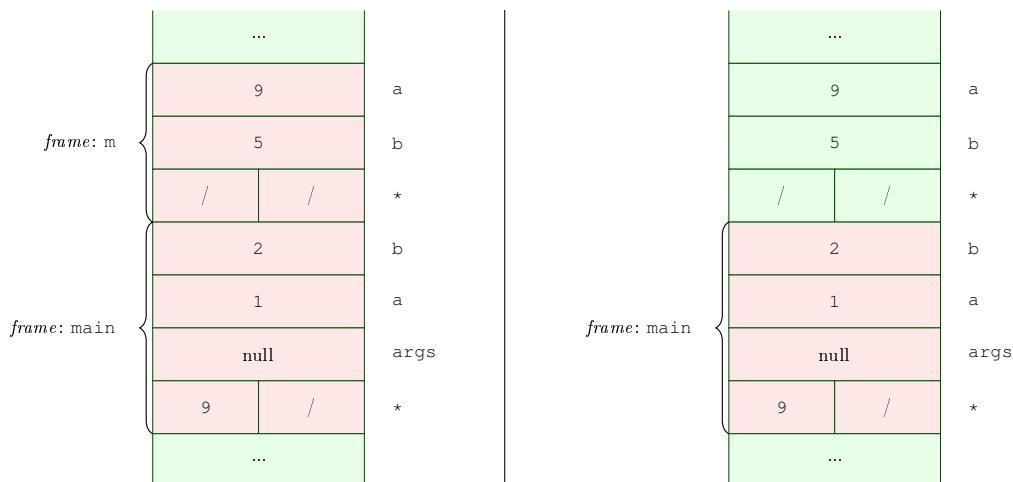
Insistiamo nel sottolineare che il *frame attivo* è sempre quello costruito per ultimo in cima al *frame stack*.

Ne consegue che il valore recuperato da b è 5 e non 2, valore associato all'occorrenza di b nel *frame* del main che non è quello *attivo*.

La somma tra 5 e 4 pari a 9 va scritta nella variabile a. Come per b, essa va cercata tra le variabili disponibili nel *frame attivo* in cima al *frame stack*, per passare dalla configurazione di sinistra a quella sulla destra:



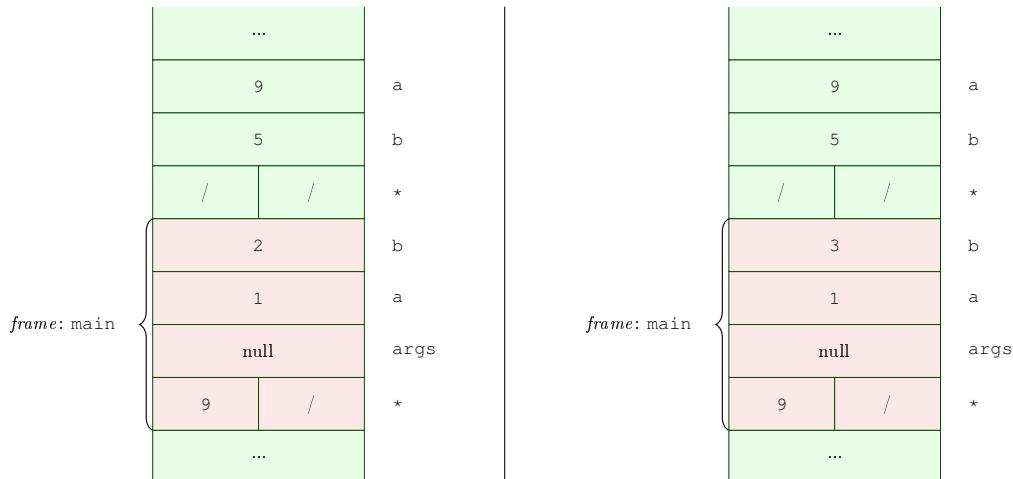
Disallocazione di m. Nessuna riga significativa segue la numero 3 di m ed il frame viene disallocato, lasciando le celle ad esso dedicate per ulteriori utilizzi, passando dalla configurazione di sinistra a quella sulla destra:



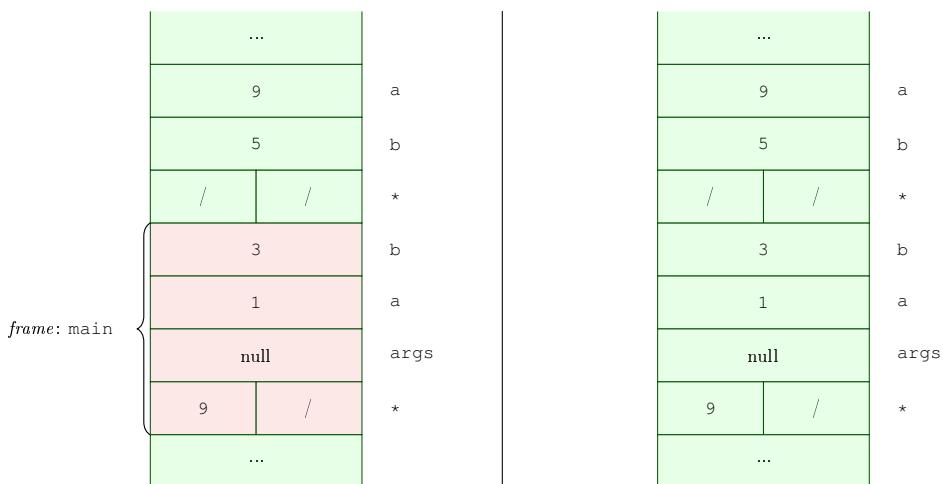
Interpretazione main (parte restante). Dopo la chiamata di m con parametro attuale 5, sua interpretazione e disallocazione del *frame*, occorre continuare l'interpretazione delle eventuali righe di codice sorgente di main. Il punto da cui continuare è noto. Esso è il numero di linea della cella * nel *frame attivo* che è quello rimasto in cima al *frame stack*, ovvero il *frame* di main.

La riga 10 richiede d'assegnare alla variabile b il valore di una espressione, usando il valore che si trova attualmente in b. La variabile b è quella che troviamo nel *frame attivo* in cima al *frame stack*: quello di main.

Quindi a b viene assegnato il valore 3, che risulta dalla valutazione di 2 + 1, passando dalla configurazione di sinistra a quella sulla destra:



Disallocazione main. Non esistendo ulteriori istruzioni significative al di sotto della riga 10, il *frame* di *main* viene disallocato, passando dalla configurazione di sinistra a quella sulla destra:



[MetodoArgomento.java](#) è il programma java che, attraverso la stampa dei valori nelle variabili, evidenzia quelle accessibili durante l'interpretazione descritta.

3.3 Metodi con parametri e risultato

Interpretiamo ora il sorgente [MetodoArgomentoRisultato.java](#):

```

1  public class MetodoArgomento {
2      public static int m (int b) {
3          int a = b + 4;
4          return a;
5      }
6
7      public static void main (String[] args) {
8          int a = 1;
9          int b = 2;
10         int c = m(b + 3) + 1;
11     }
12 }
```

cui corrisponderà l'oggetto [MetodoArgomentoRisultato.class](#) prodotto per mezzo di javac.

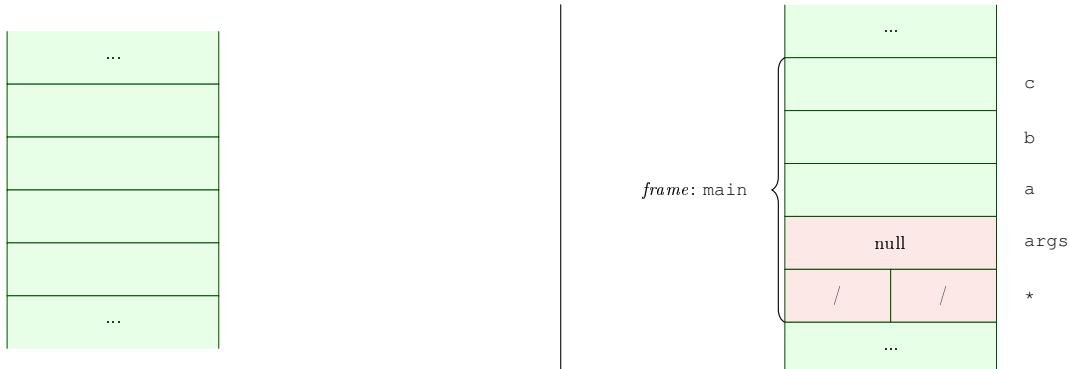
Le differenze sostanziali con [MetodoArgomentoRisultato.java](#) sono:

- nella dichiarazione `public static int m(int b)` del metodo `m` il tipo `void` è sostituito col tipo `int`. Questo significa che l'interpretazione di `m` terminerà con la restituzione al metodo chiamante di un valore di tipo `int`.
- L'ultima istruzione `return a;` di `m` è lo strumento che permette la restituzione del valore che `a` conterrà al termine dell'interpretazione del corpo del metodo `m`.

- Anche la chiamata di `m` in `main` è modificata. Essendo essa `int c = m(b + 3) + 1;`, il valore restituito dalla chiamata sarà memorizzato in `c`, dopo l'incremento di una unità.

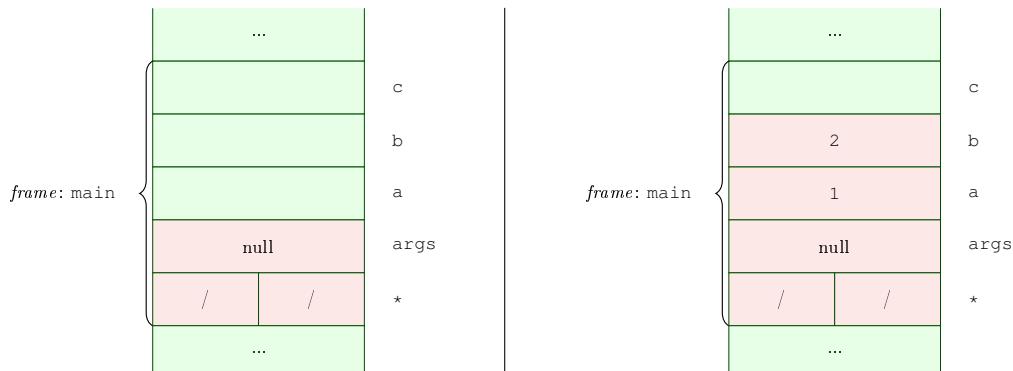
Vediamo il dettaglio su come venga gestita la memoria e di quali siano le differenze rispetto al caso precedente nel quale il metodo chiamato non restituisca alcun valore.

Allocazione main. Interpretando la riga 7, passiamo dalla configurazione a sinistra a quella destra:



Esiste la necessità di allocare spazio anche per la variabile `c`, siccome, per essa, esiste una dichiarazione.

Interpretazione main. Come in precedenza, l'interpretazione delle righe 8 e 9 permette di passare dalla configurazione di sinistra a quella sulla destra:

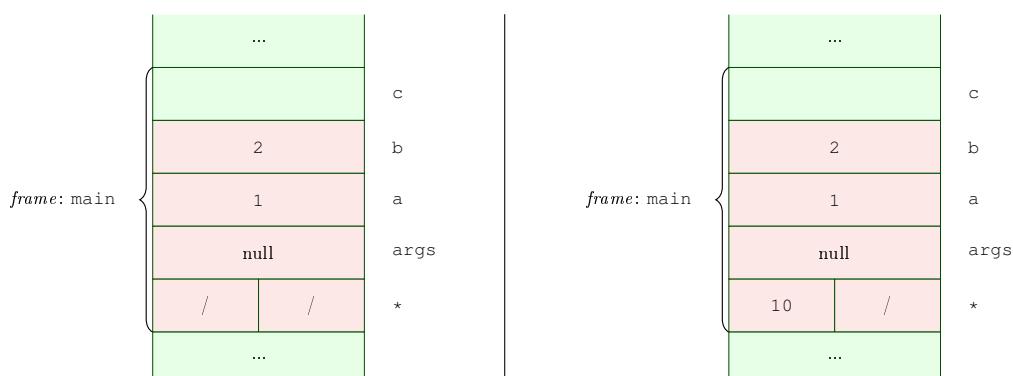


Allocazione m. L'assegnazione alla riga 10 richiede di valutare l'espressione $m(b + 3) + 1$. Il suo valore finale dipende dal quello che m , applicato al valore dell'espressione $b + 3$, produce. Nell'ordine, occorre che:

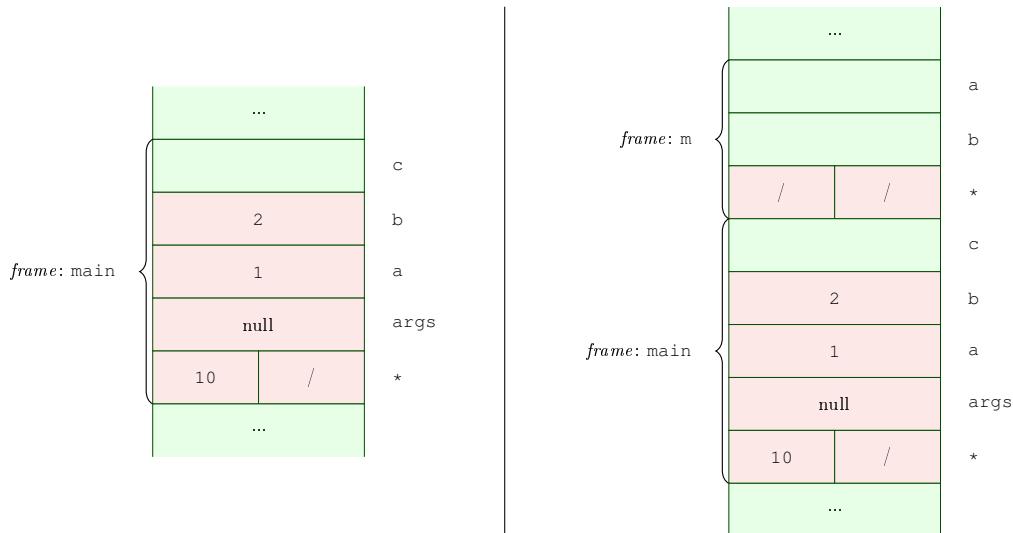
- valutiamo il valore del parametro attuale $b + 3$ da utilizzarsi per la chiamata ad m ,
 - usiamo il valore restituito dalla chiamata ad m per determinare il valore dell'intera espressione a destra dell'assegnazione alla riga 10.

Cominciamo col valutare il parametro attuale $b + 3$. Meccanicamente, come nel caso precedente, recuperiamo dal frame *attivo* il valore 2 in b , ottenendo 5 come valore per $b + 3$.

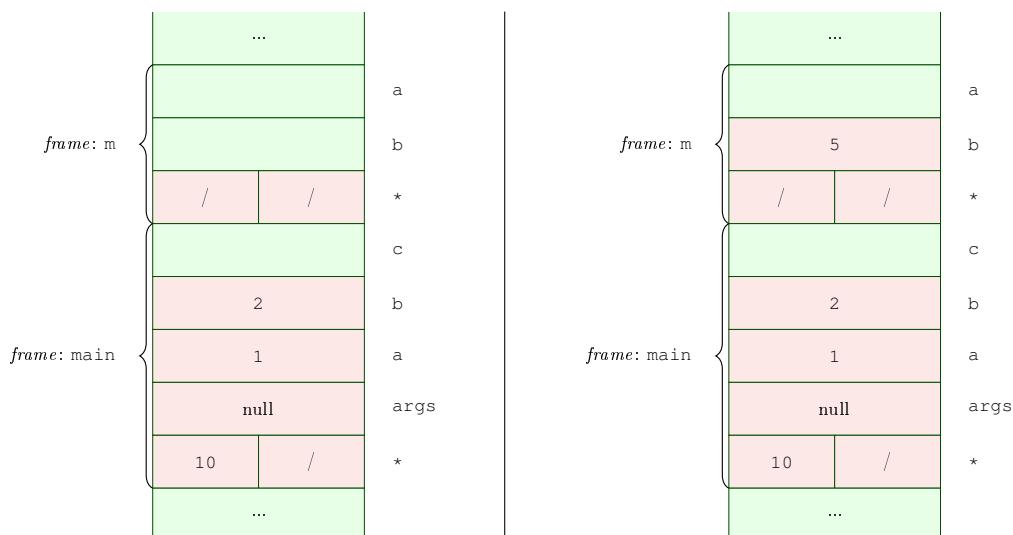
Per valutare il valore da assegnare a `c` occorre chiamare `m(b + 3)`, ovvero `m(5)`. Serve memorizzare in `*` la riga da cui occorrerà riprendere l'interpretazione del codice di `main`. Passiamo dalla configurazione di sinistra a quella sulla destra:



Allociamo il *frame* del metodo chiamato *m*, piazzandolo “sopra” il *frame* di *main* e allocando spazio per il *parametro formale b* e la *variabile locale a*. Passiamo dalla configurazione di sinistra a quella sulla destra:

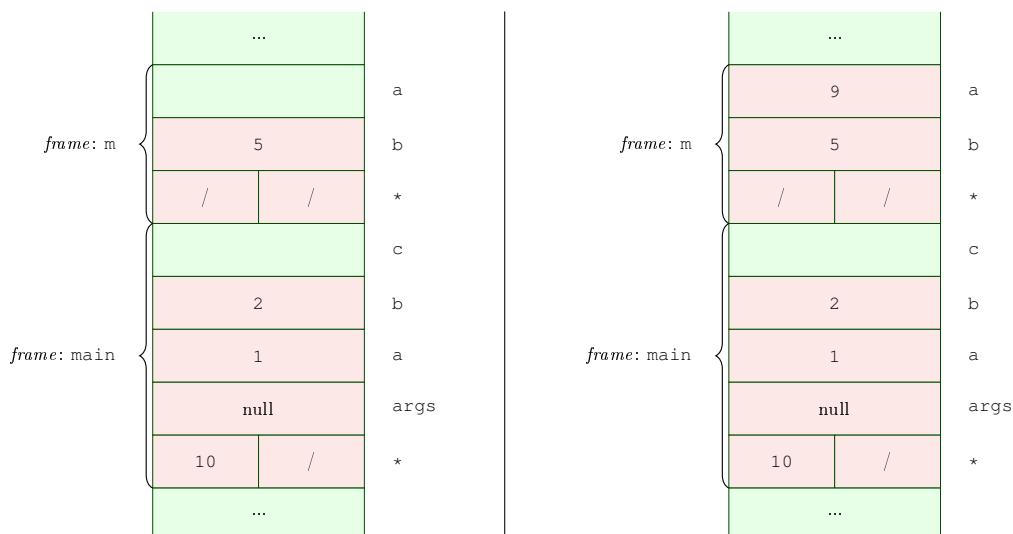


Una volta allocato il *frame* per *m* si assegna al *parametro formale* il valore del *parametro attuale* che vale 5. Passiamo dalla configurazione di sinistra a quella sulla destra:



Una volta completata l’allocazione del *frame* di *m*, lo interpretiamo.

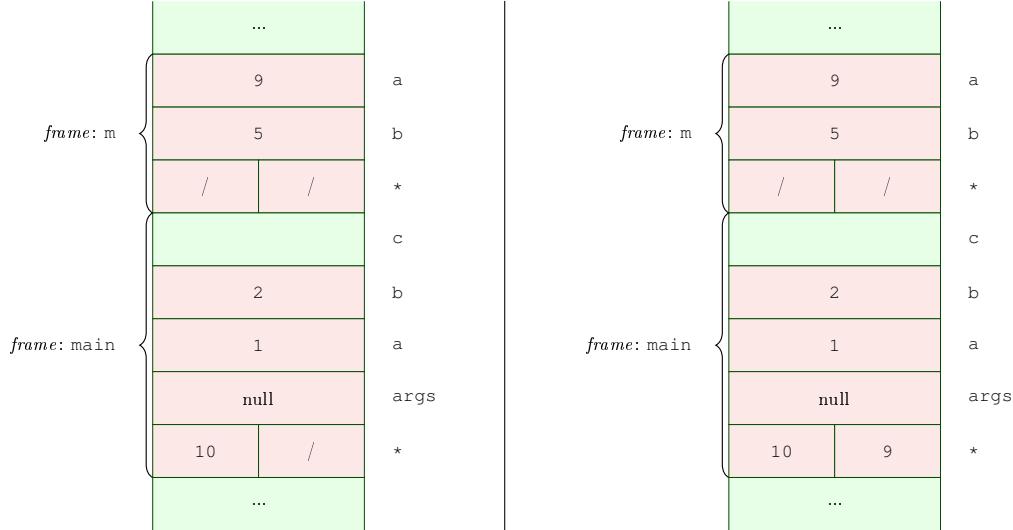
Interpretazione di *m*. L’assegnazione a riga 3 usa *b* nel *frame attivo* che contiene 5 e che, sommato con 4, permette di memorizzare 9 in *a*, sempre del *frame attivo*. Passiamo dalla configurazione di sinistra a quella sulla destra:



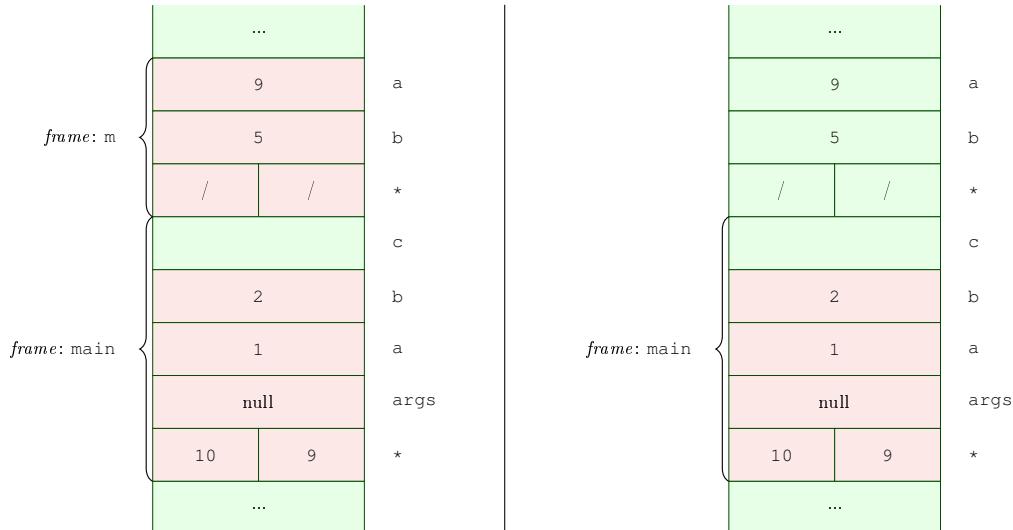
A differenza del caso precedente, occorre ancora valutare la riga 4.

Per definizione, `return <espressione>` scrive il valore di `<espressione>` nella parte non ancora utilizzata della cella di indirizzo `*` del *metodo chiamante*.

Nel nostro caso, `<espressione>` coincide con la variabile `a` del *frame attivo*. Quindi, scriviamo 9, valore di `a`, nella cella `*` del *main*, che è il *metodo chiamante*. Passiamo dalla configurazione di sinistra a quella sulla destra:



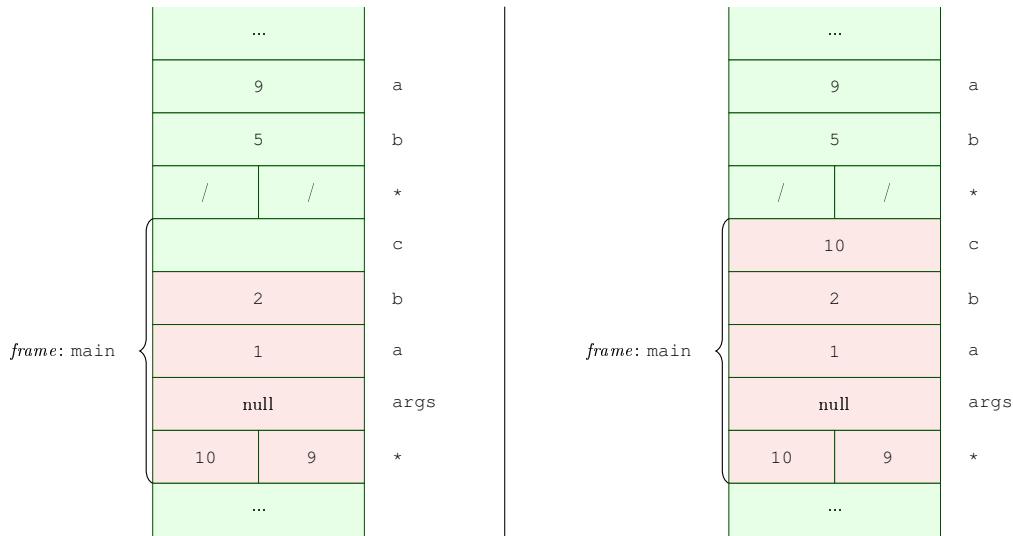
Disallocazione di m. Non esistendo più righe nel corpo di `m`, disallochiamo. Passiamo dalla configurazione di sinistra a quella sulla destra:



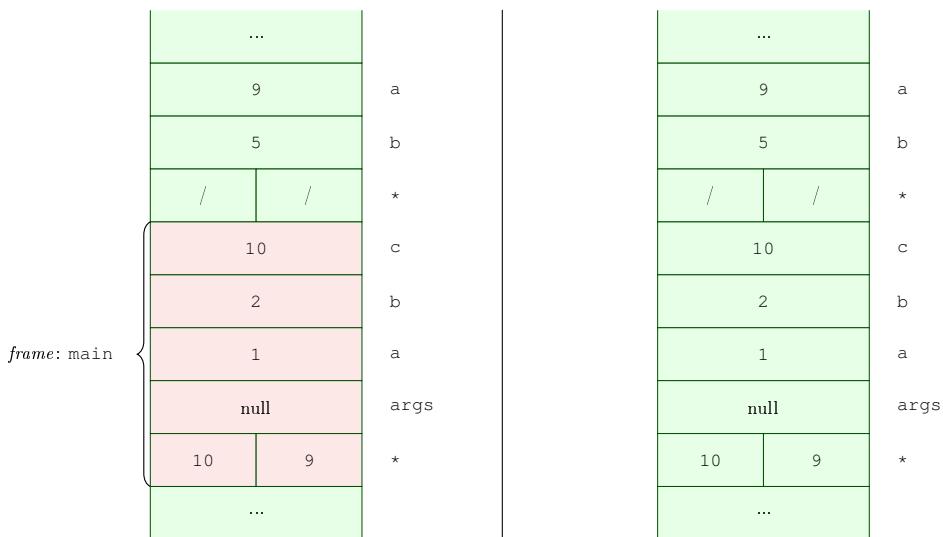
Interpretazione main (parte restante). Conclusa la chiamata di `m`, il valore da esso prodotto è recuperabile perché presente in `*` del *frame* del *main*, ora nuovamente *attivo*. Tale valore non è invece più recuperabile dal *frame* di `m` perché non più raggiungibile.

È fondamentale ricordare che la chiamata ad `m` è parte della valutazione dell'espressione a destra del simbolo `=` della riga 9 del metodo chiamante. Questo significa che occorre terminare la valutazione dell'espressione `m(b + 3) + 1;`, ovvero di `m(5) + 1`, che ora sappiamo essere il risultato della valutazione di `9 + 1`.

Concludiamo che a `c` del *frame attivo* assegniamo il valore 10. Passiamo dalla configurazione di sinistra a quella sulla destra:



Disallocazione main. Non esistendo ulteriori istruzioni significative al di sotto della riga 10, il *frame* di main viene disallocato, passando dalla configurazione di sinistra a quella sulla destra:



[MetodoArgomentoRisultato.java](#), è il programma java che, attraverso la stampa dei valori nelle variabili, evidenzi quelle accessibili durante l'interpretazione descritta.

3.4 Campi statici e final

Le variabili locali al *frame attivo* non sono le uniche disponibili per l'utilizzo da parte dei metodi. Le variabili statiche, definibili in una classe, possono essere usate come memoria condivisa cui più metodi possono accedere per leggere o per scrivere. Campi condivisi ma in sola lettura sono quelli statici di natura *final*.

Vediamo la natura dei campi statici, grazie al sorgente [VarStatiche.java](#), di cui simuliamo la gestione della memoria:

```

public class VarStatiche {
2
    static final int DELTA = 1;
    static int statica = 0;
4
6    public static void main(String[] args) {
        int x = 0;
8        mA(x);
        mB(x, 3);
10    }
12    public static void mA(int x) {
13        x = x + DELTA;
14        statica = statica + DELTA;
15    }
16}

```

```

18 public static void mB(int x, int y) {
19     final int DELTA = 5;
20     x = x + y;
21     statica = statica + DELTA;
22 }

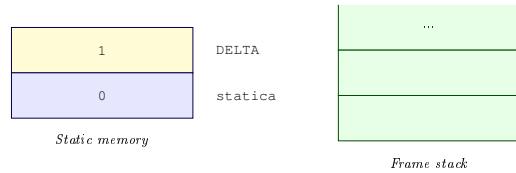
```

Rispetto a quanto visto sinora, la novità sta nella necessità di gestire una nuova zona di memoria.

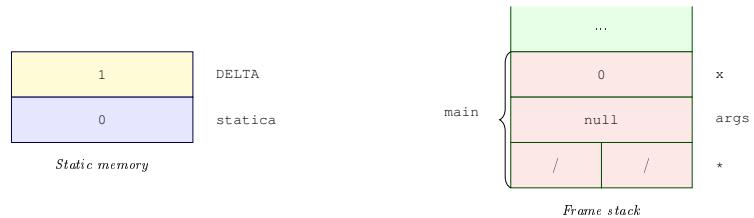
L'interpretazione della classe `VarStatiche` comincia dalle prime righe, tra le quali troviamo:

```
static final int DELTA = 1;
static int statica = 0;
```

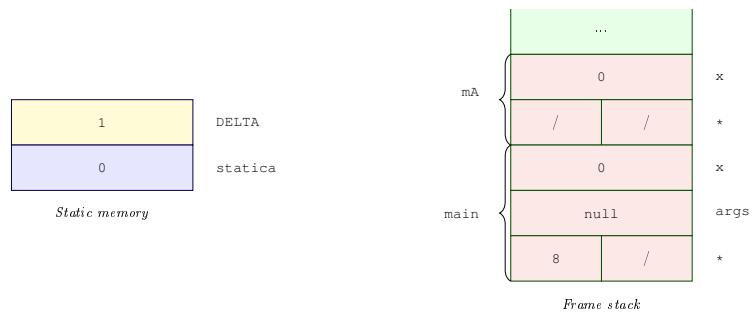
Esse allocano le due variabili `DELTA` e `statica` nella *memoria statica*, inizializzandole con i valori delle espressioni che compaiono nelle rispettive assegnazioni. Siccome il codice di nessuno dei metodi è stato interpretato, il *frame stack* è vuoto ed abbiamo la seguente situazione iniziale:



Terminata l'interpretazione del preambolo della classe, l'interprete cerca il metodo `main`. Trovandolo, lo interpreta, allocando lo spazio necessario per i parametri formali e le variabili. La situazione al termine dell'interpretazione della riga 7 diventa:



La riga 8 è un richiamo al metodo `mA` il cui parametro attuale vale 0, valore recuperato nella variabile `x` presente nel *frame attivo*. L'interpretazione della chiamata modifica la configurazione della memoria come segue:



nella quale il valore del parametro attuale è già stato assegnato al parametro formale.

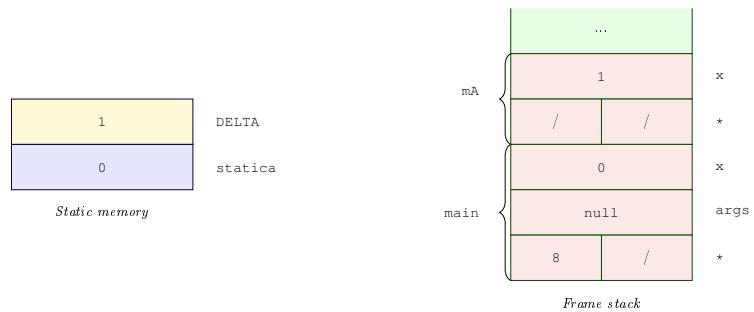
Interpretare la riga 13 richiede la valutazione dell'espressione a destra del simbolo `=` che dipende dai valori in `x` e `DELTA`. Per recuperare entrambi i valori, occorre cercare sia `x`, sia `DELTA` nel *frame attivo*.

La prima esiste nel *frame attivo* e contiene il valore 0.

La seconda non esiste nel *frame attivo*.

Quando un nome non esiste nel *frame attivo*, per definizione, esso viene cercato nella memoria statica: il valore recuperato in `DELTA` è 1.

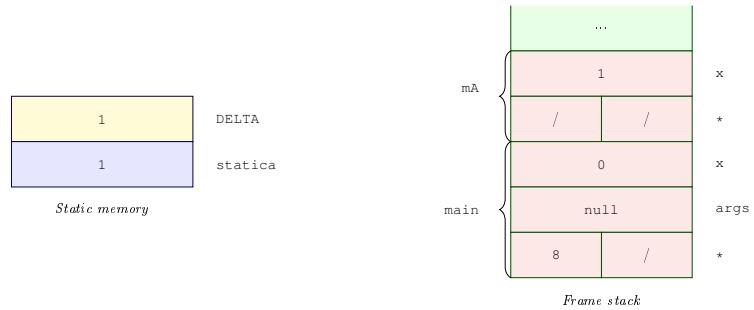
L'interpretazione della riga 13 modifica la configurazione della memoria come segue:



Interpretare la riga 14 richiede la valutazione dell'espressione a destra del simbolo = che dipende dai valori in statica e DELTA. Per recuperare entrambi i valori, occorre cercare sia statica, sia DELTA nel *frame attivo*.

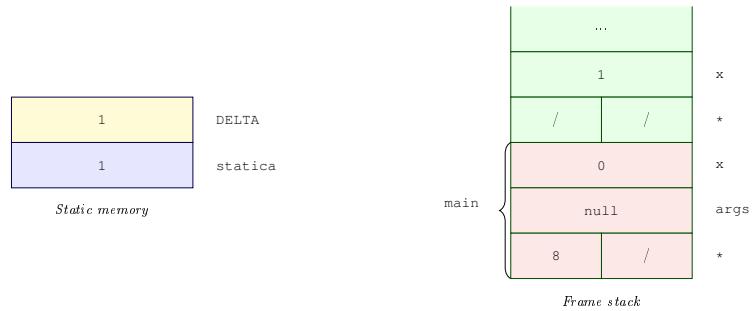
Nessuna di esse appartiene al *frame attivo*, quindi vanno cercate nella memoria statica la quale contiene entrambi i campi da cui recuperare 0 e 1, rispettivamente.

L'interpretazione della riga 14 modifica la configurazione della memoria come segue:



È fondamentale sottolineare che mA ha modificato il valore di statica, variabile che *non* appartiene al suo *frame*, ma che esiste nella memoria statica.

Il *frame attivo*, costruito per mA, in mancanza di ulteriori istruzioni da interpretare viene a disallocato:



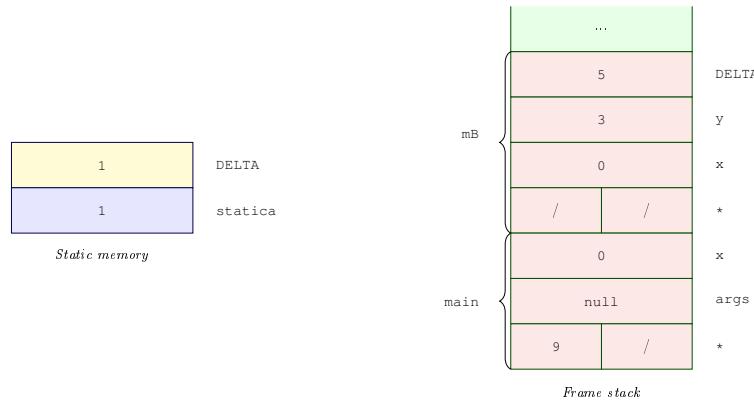
e l'interpretazione rientra alla riga indicata dalla prima componente della cella * del main, ovvero alla 9.

Siccome la riga 8 non contiene nulla se non la chiamata a mA appena interpretata, procediamo con la 9. Essa contiene la chiamata mB che richiede di valutare due parametri attuali.

La valutazione del primo restituisce il valore di x presente nel *frame attivo* del main, indipendentemente dal fatto che, in linea di principio, una variabile di nome x sia precedentemente stata allocata nel *frame* del metodo mA.

La valutazione del secondo parametro è immediata perché esso è un numero.

Conosciamo già il processo di allocazione e di assegnazione dei corretti valori al parametro formale. Otteniamo quindi, la configurazione:



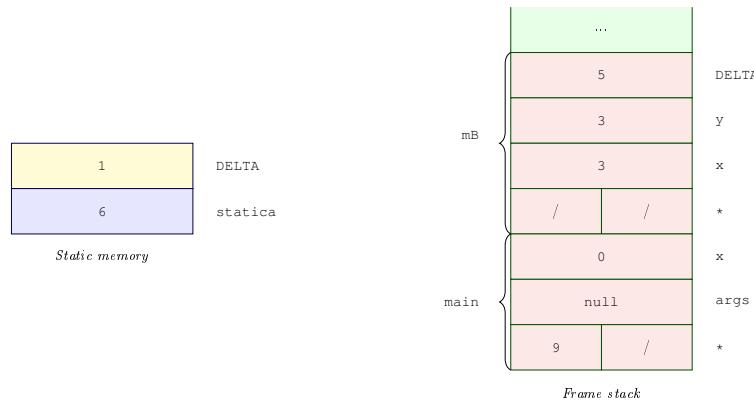
nella quale l'indirizzo della riga in cui rientrare al termine della chiamata è in * del main e l'assegnazione alla variabile final **DELTA** è già stata interpretata.

Per ipotesi, invece, non abbiamo ancora interpretato le linee 19 e 20.

La linea 19 richiede la valutazione dell'espressione **x** + **y**. Entrambi i nomi vengono cercati e trovati nel *frame attivo* di **mB**. Il risultato finale memorizzato in **x** diventa quindi 3.

La linea 20 richiede la valutazione dell'espressione **statica** + **DELTA**. Entrambi i nomi vengono cercati nel *frame attivo* di **mB**. **DELTA** vi appartiene e contiene il valore 5, distinto, in questo caso, dal valore del campo statico omonimo, presente nel preambolo della classe. Il nome **statica** esiste solo tra quelli dei campi statici e contiene il valore ottenuto dopo il precedente accesso del metodo **mA**.

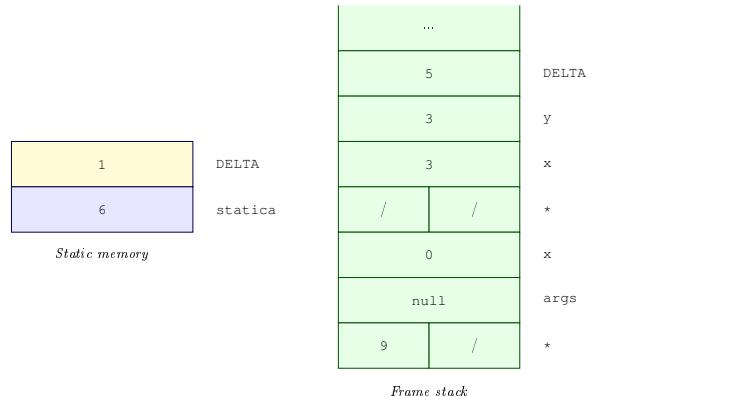
L'interpretazione di entrambe le righe produce la configurazione:



L'osservazione fondamentale è la condivisione del campo statico di nome **statica** tra i due metodi che evidenzia l'unico modo che, eventualmente, impareremo per condividere informazioni tra metodi distinti.

Terminata l'interpretazione della riga 20, il *frame* di **mB** viene disallocato, così come viene disallocato quello del **main**, subito dopo il rientro alla riga 9.

L'ultima configurazione della memoria diventa la seguente:



La nota finale è sul significato del modificatore **final**. Esso contraddistingue variabili o campi il cui valore, una volta fissato non può cambiare durante l'interpretazione per effetto di una assegnazione. Equivalentemente, se nel codice di una classe compare, ad esempio, **final costante = 1;** allora in nessun altro punto del codice di quella classe può comparire un'assegnazione **costante = <espressione>**.

`VarStatiche.java` è il programma java che, attraverso la stampa dei valori nelle variabili, evidenzia quelle accessibili durante l'interpretazione descritta.

3.5 Signature, overloading, cast, etc.

Per definizione, la *signature* di un metodo comprende nome ed elenco dei tipi dei parametri formali. L'estensione della *signature* con il tipo restituito in uscita, void incluso, si identifica come *dichiarazione*.

Il seguente sorgente `MetodoOverloading.java` sottolinea come la *signature* serva a distinguere tra le dichiarazioni dei metodi.

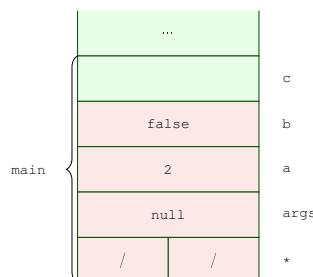
```

1  public class MetodoOverloading {
2
3      public static int m () {
4          int a = 1;
5          return a;
6      }
7
8      public static int m (int a, boolean b) {
9          int c = (b) ? a + 1 : a - 1;
10         c = c + m();
11         return c;
12     }
13
14     public static int m (float a, boolean b) {
15         int c = (int)((b) ? a + 10 : a - 10);
16         return c;
17     }
18
19     public static void main (String[] args) {
20         int a = 2;
21         boolean b = false;
22         int c = m(a + 1, true && b);
23         m(1000f, true && b);
24     }
25 }
```

Esso contiene tre dichiarazioni del metodo `m`. Non sono in conflitto perché:

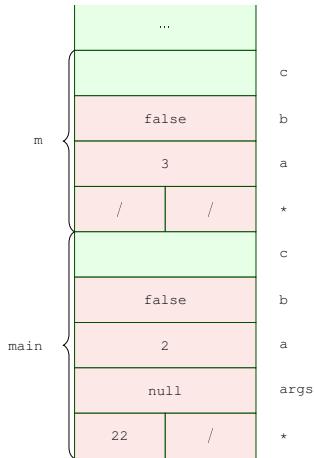
- il tipo del valore d'uscita è identico e
- prese le dichiarazioni due a due, le liste dei parametri formali differiscono o per numero di componenti, o per tipi, che occupano la stessa posizione nella lista: ad esempio, il tipo nella prima posizione della *signature* `m(int, boolean)` differisce dal tipo nella stessa posizione in `m(float, boolean)`.

L'allocazione del *frame* per il metodo `main`, e l'interpretazione delle righe 21 e 22 producono:

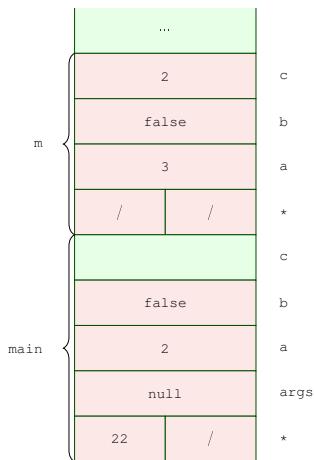


La riga 22 è un'assegnazione che contempla la valutazione di espressioni, usate come parametro attuale di `m` e la chiamata alla corretta dichiarazione di `m` tra le tre disponibili.

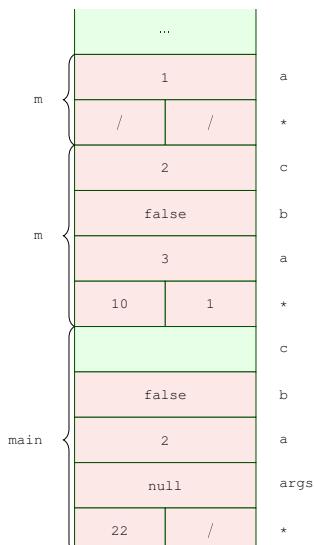
Entrambe le espressioni usano variabili locali al *frame attivo* di `main`. La prima fornisce il valore 3 e la seconda `false`. Siccome il tipo del primo parametro attuale è `int` il *frame* allocato per la chiamata è quello relativo alla dichiarazione che troviamo alla linea 8. La configurazione della memoria, ad allocazione completata, senza aver interpretato la linea 9, è la seguente:



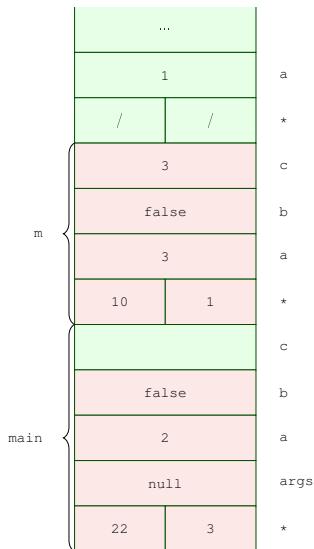
La riga 9 richiede la valutazione di una espressione condizionale `(b) ? a + 1 : a - 1`; che si interpreta, valutando b e, a seconda del suo valore, si interpreta `a + 1` o `a - 1`. Nel nostro caso, b, presente nel *frame attivo* di m, vale false. Quindi valutiamo `a - 1`, usando a presente nel *frame attivo* di m stesso. Alla fine assegniamo il valore 2 a c del *frame attivo*:



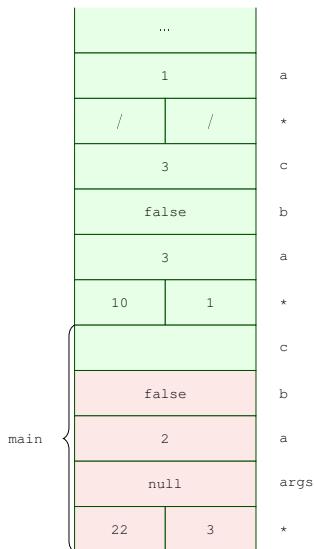
Il valore in c viene ulteriormente modificato dalla chiamata ad m nell'espressione dell'assegnazione alla riga 10. Il *frame* allocato per tale chiamata è relativo alla dichiarazione di m che troviamo alla linea 3. La configurazione della memoria immediatamente prima della disallocazione dell'ultimo *frame attivo* diventa:



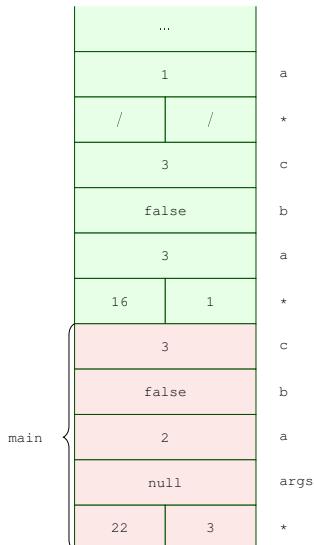
In essa vediamo che riprenderemo l'interpretazione dalla linea 10, potendo usare il valore 1 restituito dal metodo m, appena interpretato. Interpretando sino in fondo le righe 10 e 11, appena prima della disallocazione del *frame attivo*, la configurazione della memoria diventa:



Possiamo osservare che il valore di *c* nel *frame attivo* è stato scritto nella opportuna componente della cella *** del metodo chiamante *main*. Disallocando il *frame attivo* di *m* otteniamo:

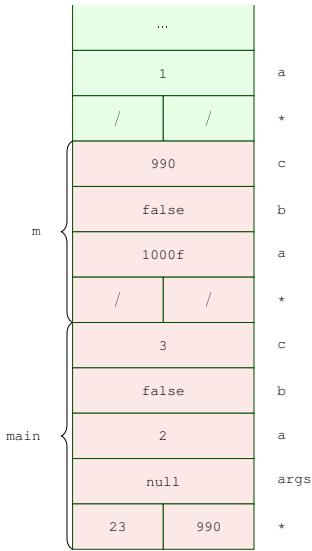


Riprendiamo dalla linea 22, come indicato dalla componente della cella *** del *frame attivo* del *main* in cui siamo appena rientrati per assegnare a *c* il valore 3 presente nella seconda componente di ***. La configurazione diventa:



Procediamo, quindi, col richiamare *m* alla linea 23. Questa volta, il primo parametro attuale è una costante il cui valore è di tipo *float*. Viene quindi allocato il frame della dichiarazione di *m* che troviamo alla riga 14.

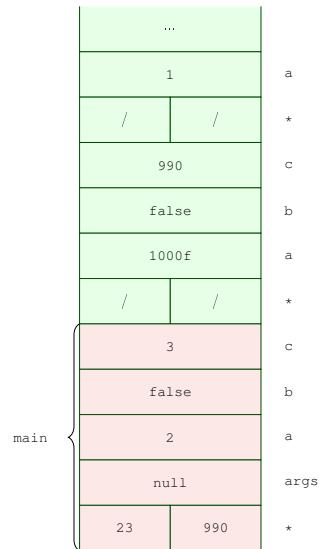
La configurazione della memoria giusto prima della disallocazione del *frame* di *m*, che segue l'interpretazione dell'istruzione *return c;* a linea 16 è:



nella quale vale la pena osservare un paio di aspetti:

- il valore di *a* è di tipo *float*, ma quello di *c* è *int*. Il motivo sta nell'aver applicato l'operazione di *cast* al risultato dell'espressione condizionale alla riga 15. Senza di essa, il compilatore segnala una possibile perdita di precisione.
- Il *frame attivo*, relativo all'ultima chiamata di *m* ha usato le celle in precedenza sfruttate per la chiamata ad una differente dichiarazione di *m*.
- Il valore 990 è comparso nella seconda componente del *frame* di *main* in cui stiamo per rientrare.

Disallocando il *frame attivo* per *m* la configurazione della memoria diventa:



Il valore 990 non viene utilizzato per alcuna assegnazione e possiamo procedere con la disallocazione finale:

...	
1	
/	/
990	
false	
1000f	
/	/
3	
false	
2	
null	
23	990

a
*
c
b
a
*
c
b
a
args
*

[MetodoOverloading.java](#) è il programma java che, attraverso la stampa dei valori nelle variabili, evidenzia quelle accessibili durante l'interpretazione descritta.

3.6 Jeliot

Un utile interprete visuale per verificare la comprensione del meccanismo di gestione della memoria da parte della JVM, ed in particolare del modo in cui il *frame stack* evolva, durante l'interpretazione di metodi ricorsivi, è [Jeliot](#) con cui, ad esempio, si può osservare la gestione della memoria durante quando si fa emergere il massimo in un array.

Capitolo 4

Elementi di base per la Correttezza parziale

Riprendiamo il problema SIDP, introdotto nella Sezione 2.3. SIDP richiede di calcolare la somma di due naturali, usando iterativamente incrementi e decrementi di opportune variabili. Supponiamo d'averne il seguente algoritmo per SIDP:

Algoritmo 7 SIDP richiamato per parlare di correttezza parziale.

```
1: // a contiene  $m \in \mathbb{N}$ 
2: // b contiene  $n \in \mathbb{N}$ 
3: while ( $b > 0$ ) do
4:   a = a + 1;
5:   b = b - 1;
6: end while
```

Cosa succederebbe se, al posto dell'Algoritmo 7, per un qualche motivo, ad esempio un banale errore di battitura, usassimo l'Algoritmo 8 come soluzione a SIDP?

Algoritmo 8 errato per SIDP

```
1: // a contiene  $m \in \mathbb{N}$ 
2: // b contiene  $n \in \mathbb{N}$ 
3: while ( $b > 0$ ) do
4:   a = a + 1;
5:   b = b - 2;
6: end while
```

La differenza sta nel decremento di due unità da b , invece che di una sola.

Una possibile ovvia strategia per rispondere alla domanda appena posta consiste nell'eseguire il *testing* dell'Algoritmo 8, ovvero consiste nel simularne il comportamento, usando valori significativi di m ed n per a e b .

Esempio 6 (Testing dell'Algoritmo 8) Assumiamo che, inizialmente, a contenga 3 e b contenga 2. Possiamo verificare che, all'uscita dall'iterazione, a valga 4 e b valga 0. Ovvero a non conterrebbe il valore desiderato.

Potremmo ripetere il *testing* per più coppie di valori in a e b . Ad un certo punto, siccome le coppie da usare per il *testing* sono infinite, giungeremmo inevitabilmente a definire la questione in termini più generali:

L'Algoritmo 8 fornirebbe un valore sbagliato in a , per una qualsiasi coppia di valori inizialmente assegnati ad a e b ?

Siccome abbiamo eseguito il *testing* anche per l'Algoritmo 3, che coincide con l'Algoritmo 7, possiamo porre su di esso la domanda complementare:

L'Algoritmo 7 fornisce il valore corretto in a , per una qualsiasi coppia di valori assegnati ad a e b ?

Per il semplice fatto che l'insieme \mathbb{N} sia infinito, è senza speranza l'immaginare di portare a termine un *testing* esaustivo né per gli Algoritmi 7 e 8, né per altri che abbiamo scritto sin qui, né per un'infinità d'altri algoritmi ben più interessanti che scrivereemo.

La domanda fondamentale diventa:

Esistono strumenti in grado di *dimostrare*, quindi in grado di *assicurare*, che, fissato un algoritmo A , per qualsiasi combinazione di valori iniziali, A restituisca il risultato voluto?

Il capitolo sviluppa gli strumenti concettuali e formali per rispondere positivamente, ovvero per parlare di *dimostrazione (verifica)* della Correttezza parziale di Algoritmi.

Gli strumenti concettuali e formali che vedremo sono la base per l'*automazione* della verifica di correttezza parziale. La verifica di correttezza parziale surclassa la qualità offerta dal *testing* sulla certificazione del buon funzionamento di un algoritmo.

4.1 Correttezza parziale all'opera

Vale la pena di cominciare citando un esempio del febbraio 2015. *Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)* è un articolo il cui succo si evince estrapolando alcune righe dei paragrafi iniziali:

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) [...]. TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. [...]

[...] Unfortunately, we weren't able to prove its correctness. A closer analysis showed that this was, quite simply, because TimSort was broken and our theoretical considerations finally led us to a path towards finding the bug (interestingly, that bug appears already in the Python implementation). [...]

Con metodi formali basati sull'individuazione di proprietà invarianti che le configurazioni manipolate da un algoritmo devono soddisfare, si è scoperto che l'algoritmo standard di ordinamento di sistemi diffusissimi conteneva un errore. In particolare, l'analisi formale delle proprietà è stata condotta usando un ambiente di sviluppo **Key**, descritto come *Integrated Deductive Software Design*. In **Key**, ad esempio, sono stati sviluppati e verificati semi-automaticamente sistemi per transazioni di pagamento elettronico, basati su carte con *chip*, sulle quali risiede software Java. La seguente figura:

Let us give a first example of this property based on the *Demoney* method **verifyPIN**. This method is present (in one form or the other) in many JAVA CARD applets, it is responsible for verifying the correctness of the PIN passed in the APDU. When the PIN is correct the method sets a global flag indicating successful PIN verification and returns. If the PIN is not correct or the maximum number of PIN entry trials has been reached an **ISOException** with a proper status code (including the number of tries left to enter the correct PIN) is thrown. Let us specify this in JML, without including the description of the status code of the exception:

— JAVA + JML —

```
/*@ public behavior
  requires apdu != null && length == DemoneyIO.VERIFY_PIN_LC
    && offset == ISO7816.OFFSET_CDATA
    && apdu._buffer != null
    && offset + length <= apdu._buffer.length
    && apdu._buffer[ISO7816.OFFSET_LC] ==
        DemoneyIO.VERIFY_PIN_LC;
  ensures true;
  signals (ISOException ie);
  signals_only ISOException;
  assignable ISOException._systemInstance._reason[0],
    pin._triesLeft[0], pin._isValidated[0];
*/
private void verifyPIN(APDU apdu, short offset, byte length)...
```

— JAVA + JML —

illustra una piccola porzione di codice, in cui sono evidenti descrizioni formali, basate sul linguaggio **Java Modeling Language**, di proprietà che i campi delle classi usate devono soddisfare e che possono essere verificate formalmente dagli strumenti disponibili in **Key**.

Uno strumento analogo a [Key](#), ma di più immediato utilizzo, è [Dafny@rise4fun from Microsoft](#) di cui illustriamo brevemente le potenzialità tramite un esempio che conosciamo bene.

Riprendiamo SIDP e consideriamo due algoritmi per esso, scritti nel linguaggio [Dafny@rise4fun from Microsoft](#):

Algoritmo 9 per SIDP in Dafny

```

1 method s(m: nat, n: nat)  {
2     var a: int := m;
3     var b: int := n;
4     while (b > 0)
5         invariant m + n == a + b;
6         invariant 0 <= b && b <= n;
7     {
8         a := a + 1;
9         b := b - 1;
10    }
11   assert m + n == a + b && b == 0;
12 }
```

Algoritmo 10 errato per SIDP in Dafny

```

1 method s(m: nat, n: nat)  {
2     var a: int := m;
3     var b: int := n;
4     while (b > 0)
5         invariant m + n == a + b;
6         invariant 0 <= b && b <= n;
7     {
8         a := a + 1;
9         b := b - 2;
10    }
11   assert m + n == a + b && b == 0;
12 }
```

Osservando la loro struttura, e cancellando idealmente quel che non conosciamo, nell'Algoritmo 9 possiamo riconoscere l'Algoritmo 7 e nell'Algoritmo 10 ritroviamo l'Algoritmo 8.

Eseguiamo due esperimenti.

Il primo consiste nel copiare il codice dell'Algoritmo 9 nella finestra apposita della pagina [Dafny@rise4fun from Microsoft](#). Un *click* sul tasto a sfondo viola produrrà, ad un certo punto, un messaggio che comunica l'avvenuta verifica senza errori. Lo stesso procedimento, ma con l'Algoritmo 10 segnalerà, in maniera un po' criptica, che il *loop invariant* non può descrivere il comportamento del *loop*, ovvero del ciclo descritto dal costrutto iterativo.

Intuitivamente, la sorgente d'errore sta nel differente valore tolto a *b*. È, invece, evidente che l'errore è scovato automaticamente. Il motivo è la presenza dei seguenti elementi sintattici, che chiamiamo *direttive*:

$$\text{invariant } m + n == a + b; \quad (4.1)$$

$$\text{invariant } 0 <= b \&& b <= n; \quad (4.2)$$

$$\text{assert } m + n == a + b \&& b == 0; \quad (4.3)$$

Lo scopo delle direttive (4.1) e (4.2) è descrivere proprietà che i valori in *m*, *n*, *a* e *b* soddisfano *sia* prima che l'assegnazione iniziale nel corpo dell'iterazione venga interpretata, *sia* appena l'ultima assegnazione nel corpo dell'iterazione venga eseguita. In particolare:

- (4.1) afferma che la somma dei valori in *m* ed *n* coincide con la somma dei valori in *a* e *b*;
- (4.2) afferma che il valore in *b* è sempre compreso tra 0 ed *n*, estremi inclusi.

Attraverso un'interpretazione passo passo del codice, sperimentiamo quanto appena affermato, riguardo a (4.1) e (4.2).

4.1.1 Introduzione dell'invariante di ciclo

Supponiamo, per esempio, che, inizialmente, nell'Algoritmo 7, *m* valga 4 ed *n* valga 3. Una volta assegnato il loro contenuto ad *a* e *b*, rispettivamente, possiamo svolgere l'interpretazione dell'iterazione come segue, finché l'espressione

$b > 0$ sia vera ed usando ancora una volta le tuple per descrivere a colpo d'occhio come i valori evolvano. Riguardo alle tuple, ci accordiamo sul fatto che la prima posizione contenga il valore di m , la seconda quello di n , la terza di a ed, infine, la quarta di b . Otteniamo la seguente successione di assegnazioni:

```
// (4, 3, 4, 3): valori all'inizio
dell'iterazione 0 con i quali abbiamo
m+n==4+3==7==4+3==a+b e b==3<=3.
a = a + 1;
b = b - 1;
// (4, 3, 5, 2): valori alla fine
dell'iterazione 0 con i quali abbiamo
m+n==4+3==7==5+2==a+b e b==2<=3.
// (4, 3, 5, 2) sono anche i valori all'inizio
dell'iterazione 1.
a = a + 1;
b = b - 1;
// (4, 3, 6, 1): valori alla fine
dell'iterazione 1 con i quali abbiamo
m+n==4+3==7==6+1==a+b e b==1<=3.
// (4, 3, 6, 1) sono anche i valori all'inizio
dell'iterazione 2.
a = a + 1;
b = b - 1;
// (4, 3, 7, 0): valori alla fine
dell'iterazione 2 con i quali abbiamo
m+n==4+3==7==7+0==a+b e b==0<=3.
```

I commenti inframezzati alle assegnazioni evidenziano come all'inizio ed alla fine di ciascun ciclo, la relazione tra i valori di m , n , a e b rimanga invariata e descrivibile col predicato $m+n==a+b$. Analogamente, b è sempre contenuta nell'intervallo indicato.

Quel che cambia, sono i valori di a e b affinché in a si *accumuli*, ciclo dopo ciclo, il valore cercato. Infine, con $b = 0$ a valore nullo, tutta la somma tra m ed n risulta accumulata in a . Questo è esattamente quanto asserisce (4.3), direttiva da interpretare strategicamente all'uscita dell'iterazione per confermare proprio che, con $b = 0$, il valore in a sia $m+n$.

L'aspetto fondamentale è che il meccanismo di mantenimento delle relazioni tra m, n, a e b vale indipendentemente dai valori che inizialmente decidiamo di assegnare ad m ed n.

Il nostro obiettivo è imparare a descrivere le relazioni tra i valori delle variabili, o delle configurazioni, che rimangano vere man mano che l'interpretazione del corpo delle iterazioni procede. Tali relazioni sono dette *predicati, o proprietà, invarianti di ciclo*.

Un invariante di ciclo, assieme alle condizioni di terminazione del ciclo in questione, permette di dimostrare che il ciclo sia in grado di produrre il risultato cercato.

Esercizio 12 • Copiare [QuozienteRestoDifferenzaIterata.dafny](#) in [Dafny@rise4fun from Microsoft](#) e ripetere esperimenti di verifica di correttezza parziale.

- Usando opportunamente le configurazioni, verificare che all'inizio ed alla fine di ogni ciclo la relazione tra n , d , q ed r sia quella indicata da $\text{invariant } n == (d * q + r);$. ■

4.2 Dimostrare che un predicato è invariante

Sinora abbiamo identificato come “invarianti” predicati dei quali, dopo qualche esperimento, riuscissimo a verificare che il valore di verità fosse vero sia all’inizio, sia alla fine di ogni iterazione in un qualche algoritmo. In realtà, un predicato può essere etichettato come “invariante” solo se *dimostriamo* che esso sia vero *per ogni* possibile valore assunto dalle componenti della configurazione di cui il predicato stesso esprime le proprietà.

4.2.1 Invariante di SIDP

Abbiamo lo scopo di dimostrare che il predicato invariante in 4.1.1 sia indipendente dai valori inizialmente assegnati a m ed n . Quindi, vogliamo che lo stesso predicato $m+n==a+b$ sia vero prima e dopo il corpo dell’iterazione. I valori in m ed n sono volutamente non specificati, perché $m+n==a+b$ deve essere vero prima e dopo il corpo dell’iterazione per *valori generici* di m ed n . Esprimendoci formalmente, vogliamo dimostrare che i predicati, espressi come commenti

Algoritmo 11 per SIDP

```

1: // m contiene un qualche numero in N
2: // n contiene un qualche numero in N
3: a = m;
4: b = n;
5: while (b > 0) do
6:   // m+n == a+b
7:   a = a + 1;
8:   b = b - 1;
9:   // m+n == a+b
10: end while

```

nell'Algoritmo 11 siano veri nel punto esatto in cui essi compaiono, utilizzando i valori delle variabili di cui essi descrivono le proprietà.

La dimostrazione consiste nel trovare manipolazioni algebriche dei valori a e b affinché lo *stesso* predicato possa essere vero sia alla linea 6, sia alla linea 9, anche se può sembrare singolare che i valori in a e b possano cambiare, passando, dalla linea 6 alla linea 9, pur avendo che il predicato $m+n == a+b$ non cambi e continui a valere.

Intuitivamente, la verità del predicato $m+n == a+b$ rimane immutata perché a e b , alla fine, sono una incrementata e l'altra decrementata della stessa quantità. Formalmente, quel che avviene è spiegato dettagliatamente nei commenti del seguente algoritmo:

```

1: // m contiene un qualche numero in N
2: // n contiene un qualche numero in N
3: a = m;
4: b = n;
5: while (b > 0) do
6:   /* Supponiamo che m+n == a+b sia vero.
7:      Allora m+n == a+b+1-1 == (a+1)+(b-1) è vero.
8:   */
9:   a = a + 1;
10:  /* L'assegnazione a = a + 1, dà il nome a al valore,
11:     presente in a prima dell'assegnazione,
12:     ma incrementato di 1.
13:     Siccome in m+n == (a+1)+(b-1) compare
14:     l'espressione a + 1, ed a è un nuovo nome per
15:     essa, possiamo rimpiazzare a + 1 con a in
16:     m+n == (a+1)+(b-1). Otteniamo così un nuovo
17:     predicato vero m+n == a+(b-1).
18:   */
19:   b = b - 1;
20:  /* L'assegnazione b = b - 1, dà il nome b al valore,
21:     presente in b prima dell'assegnazione,
22:     ma decrementato di 1.
23:     Siccome in m+n == a+(b-1) compare
24:     l'espressione b - 1, e b è un nuovo nome per
25:     essa, possiamo rimpiazzare b - 1 con b in
26:     m+n == a+(b-1). Otteniamo così un nuovo
27:     predicato vero m+n == a+b. Il nuovo predicato
28:     coincide con quello iniziale m+n == a+b.
29:     Proprio per il fatto che il predicato appena
30:     ottenuto sia identico a quello di partenza.
31:     alla linea 6, possiamo affermare che
32:     m+n == a+b sia invariante, ovvero che la sua
33:     forma sintattica non cambi da inizio a fine ciclo.
34:   */
35: end while

```

Una volta comprese le manipolazioni su a e b che permettano di riscrivere $m+n == a+b$ a linea 6 nello stesso predicato a linea 32, dovrebbe essere naturale dedurre quale sia la conseguenza di sapere che $m+n == a+b$ sia vero al termine di una qualsiasi iterazione.

Deduciamone le conseguenze.

Siccome $m+n == a+b$ è vero al termine di ogni iterazione è vero anche quando il corpo dell'iterazione non può più essere eseguito, ovvero quando $b == 0$ è vero. Essendo sia $b == 0$ sia $m+n == a+b$ veri, possiamo sostituire 0 a b in $m+n == a+b$, ottenendo $m+n == a+0 == a$. Ovvero in a , *indipendentemente* dai valori iniziali di m ed n , al termine dell'iterazione, compare il valore $m+n$, che è la somma cercata.

Il procedimento seguito è generale perché *indipendente da un qualsiasi valore specifico* fissato per m ed n . È per questo motivo che la dimostrazione di correttezza (parziale) è migliore ed indiscutibilmente più solida di un semplice *testing* per verificare che un algoritmo funzioni a dovere.

Esercizio 13 (Dimostrazioni di “invarianza”) 1. Problemi legati alla dimostrazione che i prediciati dati sono invarianti di ciclo di SIDP.

- (a) Scambiare le istruzioni 7 e 8 nell'Algoritmo 11 e dimostrare che il predicato assegnato è invariante di ciclo.
- (b) Scrivere un predicato sulle configurazioni del seguente algoritmo e dimostrare che è invariante.

```

1: // m contiene un qualche numero in N
2: // n contiene un qualche numero in N
3: a = m;
4: b = 0;
5: while (b < n) do
6:   a = a + 1;
7:   b = b + 1;
8: end while

```

- (c) Usare il predicato invariante del precedente algoritmo e verificare che non può esserlo anche per il seguente.
- ```

1: // m contiene un qualche numero in N
2: // n contiene un qualche numero in N
3: a = m;
4: b = 0;
5: while (b <= n) do
6: a = a + 1;
7: b = b + 1;
8: end while

```

2. Scrivere un predicato che descriva le proprietà delle configurazioni generate dall'iterazione nell'algoritmo che risolva MSIP: dati due numeri naturali  $m$  e  $n$ , calcolare il valore  $m*n$  come somma di  $m$  iterata per  $n$  volte. Dimostrare, quindi, che è un invariante di ciclo.

**Soluzione per MSIP.** Un metodo possibile per impostare una soluzione è osservare che  $m*n$ , vista come iterazione della somma di  $m$  per  $n$ , si può esprimere come segue:

$$m*n == \underbrace{m+\dots+m}_{n \text{ volte}} .$$

L'equazione precedente può essere riscritta come:

$$\begin{aligned} m*n &== \text{ris} + \underbrace{m+\dots+m}_{k \text{ volte}} \\ \text{ris} &== \underbrace{m+\dots+m}_{n-k \text{ volte}}, \end{aligned}$$

assumendo che  $k$  vari tra  $n$  e 0. Questa seconda coppia di equazioni indica che il risultato  $\text{ris}$  accumula via via valori ottenuti addizionando  $m$ . Il seguente algoritmo descrive il processo di accumulo dei valori in  $\text{ris}$ :

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: k = n;
5: // È facilmente verificabile che m*n==ris+m*k è vero.
6: while (k > 0) do
7: /* Supponiamo che m*n==ris+m*k sia vero.
8: Allora è vero anche m*n==(ris+m)+m*(k-1) che otteniamo dal
9: precedente, attraverso i seguenti passaggi:
10: m*n==ris+0+m*k==ris+(m-m)+m*k==(ris+m)-m+m*k==(ris+m)+m*(k-1).
11: */

```

```

12: ris = ris + m;
13: /* L'assegnazione ris = ris + m dà nome ris al valore,
14: inizialmente presente in ris prima dell'assegnazione,
15: ma incrementato di x.
16: Siccome in m*n==(ris+m)+m*(k-1) compare
17: l'espressione ris+m, e ris è un nuovo nome per
18: essa, possiamo rimpiazzare ris+m con ris in
19: m*n==(ris+m)+m*(k-1). Otteniamo così un nuovo
20: predicato vero m*n==ris+m*(k-1) in questo punto del programma.
21: */
22: k = k - 1;
23: L'assegnazione k=k-1, dà il nome k al valore,
24: presente in k prima dell'assegnazione,
25: ma decrementato di 1.
26: Siccome in m*n==ris+m*(k-1) compare
27: l'espressione k-1, e k è un nuovo nome per
28: essa, possiamo rimpiazzare k-1 con k in
29: m*n==ris+m*(k-1). Otteniamo così un nuovo
30: predicato vero m*n==ris+m*k. Il nuovo predicato
31: coincide con quello iniziale m*n==ris+m*k.
32: Proprio per il fatto che il predicato appena
33: ottenuto sia identico a quello di partenza.
34: alla linea 7, possiamo affermare che
35: m*n==ris+m*k sia invariante, ovvero che la sua
36: forma sintattica non cambi da inizio a fine ciclo.
37: */
38: end while
39: /* Per arrivare in questo punto del programma, occorre interrompere
40: l'iterazione. Questo succede se k==0. Inoltre, nessuna istruzione
41: che modifichi il valore in ris viene interpretata dopo l'ultima
42: assegnazione k=k-1 nel corpo dell'iterazione. Quindi il predicato
43: invariante m*n==ris+m*k è vero. Sostituendo 0 a k in m*n==ris+m*k
44: otteniamo m*n==ris+m*0==ris. Ovvero, al termine dell'iterazione,
45: il valore di ris è pari al prodotto di m ed n, ottenuto per somme
46: successive, ed indipendentemente dai valori iniziali di m ed n.
47: */

```

Prendendo spunto dalla dimostrazione precedente e risolvere i seguenti esercizi:

- (a) Scambiare le istruzioni 12 e 22 e procedere con la dimostrazione che il predicato dato continua ad essere un invariante di ciclo.
- (b) Identificare un predicato che possa descrivere proprietà delle configurazioni del seguente algoritmo e dimostrare che sia un invariante di ciclo.

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: k = 0;
5: while (k < n) do
6: ris = ris + m;
7: k = k + 1;
8: end while

```

- (c) Usando il predicato del punto precedente e dimostrare che non può essere un invariante di ciclo per il seguente algoritmo.

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: k = 0;
5: while (k <= n) do
6: ris = ris + m;
7: k = k + 1;
8: end while

```

3. Scrivere un predicato che descriva le proprietà dell'iterazione dell'algoritmo che risolva PMIP: dati due numeri naturali  $m$  e  $n$ , calcolare il valore  $m^n$  come moltiplicazione di  $m$  iterata per  $n$  volte. Dimostrare che il predicato è un invariante di ciclo.

**Soluzione per PMIP.** Un metodo possibile per impostare una soluzione è osservare che  $m^n$ , vista come iterazione della moltiplicazione di  $m$  per  $n$ , si può esprimere come segue:

$$m^n == \underbrace{m * \dots * m}_{n \text{ volte}} .$$

L'equazione precedente può essere riscritta come:

$$\begin{aligned} m^n &== \text{ris} * \underbrace{\dots * m}_{k \text{ volte}} \\ \text{ris} &== \underbrace{m * \dots * m}_{n-k \text{ volte}}, \end{aligned}$$

assumendo che  $k$  vari tra  $n$  e 0. Questa seconda coppia di equazioni indica che il risultato  $\text{ris}$  accumula via via valori ottenuti moltiplicando  $m$ . Il seguente algoritmo descrive il processo di accumulo dei valori in  $\text{ris}$ :

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 1;
4: k = n;
5: /* $m^n == \text{ris} * m^k$ vero, per sostituzione di valori.
6: */
7: while (k > 0) do
8: /* Suppongo $m^n == \text{ris} * m^k$ vero, riscrivibile come
9: $m^n == (\text{ris} * m) * m^{(k - 1)}$
10: */
11: ris = ris * m;
12: /* $m^n == \text{ris} * m^{(k - 1)}$ vero.
13: */
14: k = k - 1;
15: /* $m^n == \text{ris} * m^k$ vero ed identico al predicato iniziale.
16: */
17: end while
18: /* $m^n == \text{ris} * m^k \&& k = 0$
19: implica
20: $m^n == \text{ris} * m^0 == \text{ris} * 1 == \text{ris}$, ovvero
21: ris contiene il risultato, indipendentemente dai
22: valori iniziali di m ed n.
23: */

```

Prendendo spunto dalla precedente dimostrazione, risolvere i seguenti esercizi:

- (a) Scambiare le istruzioni 11 e 14 e dimostrare che il predicato individuato è ancora invariante.  
 (b) Identificare un predicato che descriva le proprietà delle configurazioni generate nel ciclo del seguente algoritmo e dimostrare che è un invariante di ciclo.

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 1;
4: k = 0;
5: while (k < n) do
6: ris = ris * x;
7: k = k + 1;
8: end while

```

- (c) Dimostrare che il predicato identificato al punto precedente non può essere invariante di ciclo del seguente algoritmo.

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: k = 0;

```

```

5: while (k < n) do
6: ris = ris * x;
7: k = k + 1;
8: end while

```

4. Definire un predicato che descriva le proprietà delle configurazioni generate dall'iterazione dell'algoritmo che risolva DPIP: dati due numeri naturali  $m$  e  $n$ , calcolare il valore  $m-n$ , iterando l'applicazione per  $n$  volte ad  $m$ . Dimostrare che il predicato è invariante.

**Soluzione per DPIP.** Un metodo possibile per impostare una soluzione è osservare che  $m-n$ , vista come iterazione di  $n$  volte del predecessore su  $m$  si può esprimere come segue:

$$m-n == m - \underbrace{1-\dots-1}_{n \text{ volte}} .$$

L'equazione precedente può essere riscritta come:

$$\begin{aligned} m-n &== \text{ris} - \underbrace{1-\dots-1}_{k \text{ volte}} \\ \text{ris} &== m - \underbrace{1-\dots-1}_{n-k \text{ volte}} , \end{aligned}$$

assumendo che  $k$  vari tra  $n$  e 0. Questa seconda coppia di equazioni indica che il risultato  $\text{ris}$  accumula via via valori ottenuti sottraendo 1. Il seguente algoritmo descrive il processo di accumulo dei valori in  $\text{ris}$ :

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: if (m > n) then
5: n = 0;
6: ris = m;
7: k = n;
8: /* m - n == ris - k è vero, per sostituzione dei valori.
9: while (k > 0) do
10: /* Suppongo m - n == ris - k vero, riscrivibile come
11: m - n == ris - k + 0 == ris - k + 1 - 1 == (ris - 1) - (k - 1).
12: */
13: ris = ris - 1;
14: /* m - n == ris - (k - 1).
15: */
16: k = k - 1;
17: /* m - n == ris - k .
18: */
19: end while
20: end if
21: /* m - n == ris - k && k == 0 implica
22: m - n == ris - k == ris - 0 == ris, ovvero ris
23: contiene il risultato cercato indifferentemente dai
24: valori iniziali di m ed n.
25: */

```

Prendendo spunto dalla dimostrazione precedente, identificare un predicato che possa essere invariante per il ciclo del seguente algoritmo:

(a)

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: k = 0;
5: if (m > n) then
6: ris = m;
7: while (k < n) do
8: ris = ris - 1;
9: k = k + 1;

```

```

10: end while
11: end if

```

5. Definire un predicato che descriva le proprietà delle configurazioni generate dall'iterazione dell'algoritmo che risolva QRDIP: dati due numeri naturali  $d$ , il divisore, ed  $s$ , il dividendo, calcolare quoziente  $q$  e resto  $r$  della divisione intera di  $d$  per mezzo di  $s$ . Dimostrare che il predicato sia invariante.

**Soluzione per QRDIP.** Dato il dividendo  $d$  ed il divisore  $s$  il predicato che esprime la relazione tra quoziente  $q$  e resto  $r$ , usando sottrazioni successive di  $s$  da  $d$ , avrà la forma seguente:

$$d == \underbrace{s + \dots + s}_{q \text{ volte}} + r \text{ con } r < s .$$

Il seguente algoritmo descrive il processo di accumulo dei valori in  $q$  ed  $r$ :

```

1: /* // d contiene un valore di N
2: /* // s contiene un valore di N
3: r = d;
4: q = 0;
5: /* // d == q*s + r vero per sostituzione di valori
6: while (r >= s) do
7: /* Suppongo d == q*s + r vero, riscrivibile come
8: d == q*s + 0 + r == q*s + s - s + r == (q + 1)*s + (r - s).
9: */
10: r = r - s;
11: /* d == (q + 1)*s + r
12: */
13: q = q + 1;
14: /* d == q*s + r
15: */
16: end while
17: /* d == q*s + r && r < s è il predicato che vale
18: in questo punto del codice e coincide con la relazione
19: tra d, s, q, ed r che cerchiamo.
20: */

```

Usando la dimostrazione precedente come spunto e scambiando le istruzioni 10 e 13, dimostrare che il predicato rimane invariante di ciclo. ■

#### 4.2.2 Correttezza parziale per QPNP

Per definizione, chiamiamo QPNP il problema che richieda di calcolare il quadrato di un numero naturale  $x$ , sfruttando la seguente specifica proprietà dei prodotti notevoli: il valore  $(x+1) * (x+1)$  si può esprimere in termini del valore di  $x*x$ , ovvero esiste una relazione *ricorsiva* tra  $(x+1)^2$  e  $x^2$ . Vedremo in seguito la rilevanza del concetto “ricorsione”.

Segue l'algoritmo dai cui commenti è possibile ricavare la relazione tra  $(x+1)^2$  e  $x^2$ :

```

1: // x contiene un valore di N
2: n = 0;
3: ris = 0;
4: while (n < x) do
5: /* n*n == ris
6: implica
7: n*n+2*n+1 == ris+2*n+1 == ris+(n+1)+(n+1)-1
8: implica
9: (n+1)*(n+1) == ris+2*(n+1)-1
10: */
11: n = n + 1;
12: /* n*n == ris+2*n-1
13: */
14: ris = ris + 2 * n - 1;
15: /* n*n == ris && n <= x
16: implica
17: n*n == ris
18: */

```

```

19: end while
20: /* n*n == ris && n = x
21: implica
22: x*x == ris
23: */

```

**Esercizio 14 (Correttezza parziale legata a QPNP)** • Qual è il predicato invariante del seguente algoritmo, analogo al precedente, ma con assegnazioni scambiate tra loro?

```

1: // x contiene un valore di N
2: n = 0;
3: ris = 0;
4: while (n < x) do
5: ris = ris + 2 * n + 1;
6: n = n + 1;
7: end while
8: */

```

- Procedendo in maniera analoga a quanto fatto per QPNP, usare il prodotto notevole appropriato per identificare l'invariante di ciclo per l'algoritmo che calcoli il cubo di un valore numerico. Il vincolo per scrivere l'algoritmo è saper calcolare solo somme, elevamenti al quadrato e triplicazioni.

#### 4.2.3 Predicati per selezioni

Nella sezione precedente abbiamo insistito sui predicati invarianti di ciclo e su come la loro struttura sintattica rimanga invariata nel descrivere sia la configurazione che precede la prima istruzione del corpo di una iterazione sia quella che segue l'ultima istruzione dello stesso corpo.

È interessante “spargere” predicati che descrivono proprietà delle configurazioni anche in altri punti di un algoritmo, oltre che all’inizio ed alla fine del corpo di una iterazione.

Il seguente algoritmo per il problema SelP illustra come predicati possano descrivere quel che succede alle configurazioni in presenza di selezioni:

```

1: // a contiene m ∈ N
2: // b contiene n ∈ N
3: // Per ipotesi, la configurazione si descrive col predicato a == m && b == n
4: if (a > b) then
5: /* Percorrere questo ramo, permette di meglio specializzare
6: la proprieta' della configurazione, aggiungendo ad essa il fatto
7: che la condizione della selezione sia vera:
8: a == m && b == n && a > b . (1)
9: Qualsiasi istruzione seguente agira' su configurazioni in
10: cui vale (1) e qualiasi predicato che (1) possa implicare.
11: Ad esempio, (1) implica:
12: a-b == m-n , (2)
13: ottenuta applicando ad a==m una sottrazione ad entrambi
14: i membri di quantita' identiche fra loro, cioe' b ed n.
15: */
16: a = a - b;
17: /* L'assegnazione chiama col nome a il valore
18: della sottrazione del valore in b da quello di a.
19: Come in altre situazioni analoghe, sostituiamo a ad
20: a-b in (2), ottenendo almeno:
21: a == m-n . (3)
22: Siccome, prima dell'assegnazione è vero che a > b, avremo:
23: a > 0 . (4)
24: Al termine di questo ramo della selezione le
25: proprieta' rilevanti dello stato sono descritte
26: dal predicato:
27: a == m-n && a > 0 . (5)
28: */
29: else
30: /* Percorrere questo ramo, permette di meglio specializzare
31: la proprieta' della configurazione, aggiungendo ad essa il fatto
32: che la condizione del costrutto condizionale sia vera:

```

```

33: a == m && b == n && a <= b . (6)
34: Qualsiasi istruzione seguente agira' su configurazioni in
35: cui vale (6) e qualiasi predicato che (6) possa
36: implicare.
37: Ad esempio, (6) implica:
38: b-a == n-m, (7)
39: ottenuta applicando a b==n una sottrazione ad entrambi
40: i membri di quantita' identiche fra loro, cioe' a ed m.
41: */
42: a = b - a;
43: /* L'assegnazione chiama col nome a il valore
44: della sottrazione del valore di a da quello di b. E'
45: quindi possibile sostituire a al posto di
46: b-a in (7), ottenendo almeno:
47: a == n-m . (8)
48: Siccome , prima dell'assegnazione a<=b, avremo:
49: a >= 0 . (9)
50: Al termine di questo ramo della selezione le
51: proprietà rilevanti dello stato sono descritte dal
52: predicato:
53: a == n-m && a >= 0 . (10)
54: */
55: end if
56: /* In questo punto del programma abbiamo percorso uno dei due rami
57: quindi, grazie a (5) e (10) l'unica cosa che possiamo sapere è:
58: a >= 0 .
59: */

```

#### 4.2.4 Invariante per il problema MCD

MCD è il classico problema di calcolare il massimo comun divisore di due numeri naturali  $x$  ed  $y$ . C'è chi (Euclide ...) ha studiato per noi le proprietà del MCD. Esse sono riassumibili attraverso il seguente insieme di equazioni:

$$\text{MCD}(x, y) = \begin{cases} \text{MCD}(x - y, y) & \text{se } x > y \\ \text{MCD}(x, y - x) & \text{se } x < y \\ x & \text{se } x = y \end{cases} . \quad (4.4)$$

Le equivalenze espresse in (4.4) suggeriscono una lettura algoritmica per ricavare  $\text{MCD}(x, y)$ : finché  $x$  ed  $y$  sono diversi decrementiamo l'argomento maggiore e continuiamo a cercare il valore MCD tra l'argomento appena diminuito di valore e l'altro, lasciato invariato.

Il seguente algoritmo, ed i predicati, formalizzano quanto descritto:

```

1: // a contiene x ∈ N
2: // b contiene y ∈ N
3: // MCD(a,b) == MCD(x,y)
4: while (a != b) do
5: /* a!=b && MCD(a,b)==MCD(x,y) implica
6: MCD(a,b)==MCD(x,y) && (a > b || a < b)
7: */
8: if (a > b) then
9: /* a > b implica MCD(a-b,b)==MCD(a,b)==MCD(x,y)
10: */
11: a = a - b;
12: // MCD(a,b) == MCD(x,y)
13: end if
14: if (b > a) then
15: /* b > a implica MCD(a,b-a)==MCD(a,b)==MCD(x,y)
16: */
17: b = b - a;
18: // MCD(a,b) == MCD(x,y)
19: end if
20: end while

```

```

21: /* a==b && MCD(a,b)==MCD(x,y) implica
22: MCD(a,a)==MCD(x,y) che, assieme al terzo assioma a==MCD(a,a), implica a==MCD(x,y)
23: */

```

**Esercizio 15 (Invariante per MCD alternativo)** Scrivere il predicato invariante per le iterazioni nel seguente algoritmo, alternativo al precedente, per l'MCD tra numeri naturali  $x$  ed  $y$ :

```

1: // a contiene x ∈ N
2: // b contiene y ∈ N
3: while (a != b) do
4: while (a > b) do
5: a = a - b;
6: end while
7: while (a < b) do
8: b = b - a;
9: end while
10: end while

```

La soluzione è in [MCDTriploCiclo.java](#).

## 4.3 Il principio di induzione

Prima di procedere oltre, è utile richiamare principio di induzione che vedremo strettamente collegato sia alla dimostrazione di correttezza parziale per riscrittura del predicato (Sezione 4.2), sia all'approccio *ricorsivo* nella progettazione di algoritmi (Capitolo 5).

Nell'articolo di wikipedia sul [principio d'induzione](#) si trova un'introduzione divulgativa al concetto di induzione del quale segnaliamo l'esempio iniziale sulle tessere del domino, interessante perché sembra sottolineare l'aspetto dinamico che il principio di induzione sembra sottendere. Qui di seguito riassumiamo i punti essenziali per le applicazioni nell'ambito della programmazione.

Una trattazione più completa verrà fatta nel corso di Matematica discreta e Logica.

In matematica (ed in informatica) è spesso necessario dimostrare che una certa proprietà è vera per tutti i numeri naturali. Alcuni esempi:

- Per ogni  $n \in \mathbb{N}$ ,

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}.$$

- Consideriamo un frammento di codice della forma:

```

while (b)
 S;

```

Se  $P$  è una proposizione che esprime una relazione tra i valori delle variabili che compaiono nell'istruzione  $S$ , allora si può definire un'altra proprietà

$Q(n) \leftrightarrow P$  è vera dopo  $n$  iterazioni del ciclo while.

Proprietà di questo tipo sono utilizzate per stabilire che  $P$  è una proprietà invariante del ciclo in questione.

La formulazione più generalmente nota del principio di induzione è la seguente:

**PRINCIPIO DI DEMOSTRAZIONE PER INDUZIONE (PI):** Data una proprietà  $P$  dei numeri naturali, se  $P(0)$  e  $P(n) \Rightarrow P(n+1)$ , allora  $\forall x \in \mathbb{N}.P(x)$ .

Qui una proprietà dei numeri naturali è una proprietà per la quale abbia senso chiedersi se è vera o falsa per un numero naturale. La **BASE** dell'induzione è la dimostrazione di  $P(0)$ , mentre il **PASSO INDUTTIVO** è la dimostrazione dell'implicazione  $P(n) \Rightarrow P(n+1)$  che, normalmente, si articola nel modo seguente: si assume che  $P(n)$  sia vera (questa è detta l'**IPOTESI INDUTTIVA**) e si dimostra che  $P(n+1)$ . Un'altra formulazione, del tutto equivalente alla prima, del principio di induzione, usa l'**ESTENSIONE** della proprietà  $P$ , cioè l'insieme dei numeri naturali per i quali la proprietà è vera:

Se  $A \subseteq \mathbb{N}$  è tale che  $0 \in A$  e, per ogni  $n \in \mathbb{N}$ ,  $n \in A \Rightarrow n+1 \in A$ , allora  $A = \mathbb{N}$ .

**Aritmetica** Vediamo subito il primo esempio di utilizzo del principio di induzione:

**Teorema 1**

Per ogni  $n \in \mathbb{N}$ ,

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}.$$

**Dim. 1** Qui la proprietà  $P(k)$  è

$$\sum_{i=0}^k i = \frac{k(k+1)}{2}.$$

La base consiste nel verificare che entrambi i lati dell'induzione hanno valore 0. Per dimostrare il passo induttivo, assumiamo che

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}$$

e dimostriamo che

$$\sum_{i=0}^{n+1} i = \frac{(n+1)(n+2)}{2}.$$

Ora,

$$\begin{aligned} \sum_{i=0}^{n+1} i &= \left( \sum_{i=0}^n i \right) + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \quad \text{per l'ipotesi induttiva} \\ &= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

e mediante un'applicazione del principio di induzione si ottiene la conclusione.

**Teorema 2**

Per ogni  $m \in \mathbb{N}$ :

$$(m+1)^2 = (m+1) + 2 \sum_{i=0}^{m-1} (m-i) .$$

**Dim. 2** Qui la proprietà  $P(m)$  è:

$$(m+1)^2 = (m+1) + 2 \sum_{i=0}^m (m-i) .$$

La base consiste nel verificare che entrambi i lati dell'equazione siano identici quando  $m = 0$ :

$$(0+1)^2 = 1 = (0+1) + 2 \sum_{i=0}^{0-1} (m-i) .$$

Per dimostrare il passo induttivo, assumiamo che:

$$(m+1)^2 = (m+1) + 2 \sum_{i=0}^{m-1} (m-i) \quad (4.5)$$

e dimostriamo che:

$$(m+2)^2 = (m+2) + 2 \sum_{i=0}^m (m+1-i) .$$

Cominciamo con l'espandere l'espressione a sinistra e col manipolare la sommatoria, con l'obiettivo di far comparire esplicitamente (4.5):

$$\begin{aligned}
 (m+1)^2 + 2(m+1) + 1 &= (m+2) + 2 \sum_{i=0}^m (m+1-i) \\
 &= (m+2) + 2 \sum_{i=0}^m (m-i+1) \\
 &= (m+2) + 2(m-m+1) + 2 \sum_{i=0}^{m-1} (m-i+1) \\
 &= (m+2) + 2 + 2 \sum_{i=0}^{m-1} (m-i) + 2 \sum_{i=0}^{m-1} 1 \\
 &= (m+2) + 2 + 2 \sum_{i=0}^{m-1} (m-i) + 2m .
 \end{aligned}$$

Isolando  $(m+1)^2$  a sinistra, otteniamo:

$$\begin{aligned}
 (m+1)^2 &= -2m - 2 - 1 + m + 2 + 2 + 2m + 2 \sum_{i=0}^{m-1} (m-i) \\
 &= (m+1) + 2 \sum_{i=0}^{m-1} (m-i) ,
 \end{aligned}$$

che è proprio (4.5), vera per ipotesi.

**Esercizio 16** Dimostrare mediante induzione che, per ogni  $n > 0$ ,  $n^3 - n$  è divisibile per 3.

**Esercizio 17** Dimostrare mediante induzione che, per ogni  $n \geq 4$ ,  $n! > 2^n$ .

**Esercizio 18** Dimostrare mediante induzione che, per ogni  $n > 0$ ,

$$\frac{n(3n-1)}{2} = \sum_{i=1}^n (3i-2).$$

**Esercizio 19** Dimostrare mediante induzione che, per ogni  $n \geq 3$ ,  $n^2 > 2n + 1$ .

**Esercizio 20** Dimostrare mediante induzione che, per ogni  $n \geq 5$ ,  $2^n > n^2$ .

[elearning.math.unipd.it](http://elearning.math.unipd.it) contiene alcune soluzioni esplicite agli esercizi precedenti.

## 4.4 Dimostrazione della Correttezza parziale per induzione

Questa sezione è focalizzata sulla dimostrazione della correttezza parziale in presenza di una iterazione. In tal caso, la dimostrazione procede per induzione sul numero di esecuzioni complete del corpo dell'iterazione, usando la dimostrazione che un predicato sia invariante di ciclo per dimostrare il passo induttivo.

### 4.4.1 Correttezza parziale di SIDP

Dato:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = m;
4: i = n;
5: while (i > 0) do
6: ris = ris + 1;
7: i = i - 1;
8: end while

```

vogliamo dimostrare che, se il ciclo termina, allora la variabile `ris` contiene un valore pari a  $m+n$ .

La dimostrazione procede per induzione sul numero  $k$  di iterazioni, ovvero, sul numero di volte che l'intero corpo dell'iterazione:

```
ris = ris + 1;
i = i - 1;
```

viene percorso.

Osserviamo che le variabili `ris` e `i` cambiano entrambe valore al termine di ogni ciclo. Ad esempio, dopo  $k$  cicli, per  $k$  opportuno, il valore di `ris` è diverso dal valore assunto dopo  $k+1$  cicli. Questo suggerisce di identificare come segue il valore di `ris` e `i` in funzione del numero  $0 \leq k$  di cicli percorsi:

- denotiamo con  $ris_k$  il valore della variabile `ris` al termine del  $k$ -esimo ciclo,
- denotiamo con  $i_k$  il valore della variabile `i` al termine del  $k$ -esimo ciclo.

Lo scopo è dimostrare:

### Proprietà 1

Siano  $m$  e  $n$  fissati. Per ogni  $0 \leq k \leq n$ , se la sequenza di istruzioni 6 e 7 è stata percorsa  $k$  volte, allora  $ris_k + i_k == m+n$  e  $i_k == (i-k)$ .

### Dim.

- Supponiamo  $k = 0$ . Il codice contiene le assegnazioni `ris=m` e `i=n` che sono eseguite prima di una qualsiasi iterazione. Quindi,  $ris_0 == m$  e  $i_0 == n$  e l'enunciato vale banalmente:  $ris_0 + i_0 == m+n$  e  $i_0 == (i-0)$ .
- Dato  $0 < k < n$ , per ipotesi induttiva assumiamo  $ris_k + i_k == m+n$  e  $i_k == (i-k)$ . La prima equivalenza vale grazie ai seguenti passi:

$$\begin{aligned} ris_{k+1} + i_{k+1} &= ris_k + i_k + 1 && \text{(istruzione 4)} \\ &= ris_k + i_k - 1 && \text{(istruzione 5)} \\ &= ris_k + i_k \\ &= m+n && \text{(ipotesi induttiva)} . \end{aligned}$$

La seconda equivalenza vale grazie ai seguenti passi:

$$\begin{aligned} i_{k+1} &= i_k + 1 && \text{(istruzione 4)} \\ &= (i-k) + 1 && \text{(ipotesi induttiva)} \\ &= i - (k+1) . \end{aligned}$$

□

### Corollario 1

Siano  $m$  e  $n$  fissati. Dopo  $n$  cicli,  $m+n == ris$ .

### Dim.

$$\begin{aligned} m+n &= ris_n + i_n && \text{(Proprietà 1 e } k==n\text{)} \\ &= ris_n + 0 && \text{(Proprietà 1)} \\ &= ris_n \\ &= ris . \end{aligned}$$

□

**Esercizio 21 (Correttezza parziale per induzione con soluzioni)** 1. Per induzione, dimostrare la correttezza parziale del problema MSIP che, dati due numeri naturali  $m$  e  $n$ , impone di calcolare il valore  $m*n$  come somma di  $m$  iterata per  $n$  volte.

**Soluzione per MSIP.** Dato:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: i = n;
5: while (i > 0) do
6: ris = ris + m;
7: i = i - 1;
8: end while

```

vogliamo dimostrare che, se il ciclo termina, allora la variabile `ris` contiene un valore pari a  $x * y$ .

La dimostrazione procede per induzione sul numero  $k$  di iterazioni, ovvero, sul numero di volte che l'intero corpo dell'iterazione:

```

ris = ris + x;
i = i - 1;

```

viene percorso.

Osserviamo che le variabili `ris` e `i` cambiano entrambe valore al termine di ogni ciclo. Ad esempio, dopo  $k$  cicli, per  $k$  opportuno, il valore di `ris` è diverso dal valore assunto dopo  $k+1$  cicli. Questo suggerisce di identificare il valore di `ris` e `i` in funzione del numero  $0 \leq k$  di cicli percorsi:

- denotiamo con  $risk_k$  il valore della variabile `ris` al termine del  $k$ -esimo ciclo,
- denotiamo con  $i_k$  il valore della variabile `i` al termine del  $k$ -esimo ciclo.

### Proprietà 2

Siano  $m$  e  $n$  fissati. Per ogni  $0 \leq k \leq n$ , se la sequenza di istruzioni 6 e 7 è stata percorsa  $k$  volte, allora  $risk_k + (i_k * m) == m * n$  e  $i_k == (n - k)$ .

#### Dim.

- Supponiamo  $k = 0$ . Il codice contiene le assegnazioni `ris=0` e `i=n` che sono eseguite prima di una qualsiasi iterazione. Quindi,  $risk_0 == 0$  e  $i_0 == n$  e l'enunciato vale banalmente:  $risk_0 + (i_0 * m) == m * n$  e  $i_0 == (n - 0)$ .
- Dato  $0 < k < n$ , per ipotesi induttiva assumiamo  $risk_k + (i_k * m) == m * n$  e  $i_k == (n - k)$ . La prima equivalenza vale grazie ai seguenti passi:

$$\begin{aligned}
risk_{k+1} + (n_{k+1} * m) &== risk_k + m + (i_{k+1} * m) && \text{(istruzione 4)} \\
&== risk_k + m + ((i_k - 1) * m) && \text{(istruzione 5)} \\
&== risk_k + m + (i_k * m) - m \\
&== risk_k + i_k \\
&== m * n && \text{(ipotesi induttiva)} .
\end{aligned}$$

La seconda equivalenza vale grazie ai seguenti passi:

$$\begin{aligned}
i_{k+1} &== i_k + 1 && \text{(istruzione 4)} \\
&== (i - k) + 1 && \text{(ipotesi induttiva)} \\
&== i - (k + 1) .
\end{aligned}$$

□

### Corollario 2

Siano  $m$  e  $n$  fissati. Dopo  $n$  cicli,  $m * n == ris$ .

#### Dim.

$$\begin{aligned}
m * n &== risk_k + i_k * m && \text{(Proprietà 2 e } k == n) \\
&== ris_n + n_y * m && \text{(Proprietà 2)} \\
&== ris_n + 0 * m \\
&== ris_n \\
&== ris .
\end{aligned}$$

□

Prendendo spunto dalla dimostrazione di correttezza parziale per induzione di MSIP, risolvere i seguenti esercizi:

(a) Scambiare le istruzioni 6 e 7 e dimostrare la correttezza parziale.

(b) Dimostrare la correttezza parziale de:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: i = 0;
5: while (i < n) do
6: ris = ris + m;
7: i = i + 1;
8: end while

```

(c) Dimostrare che il seguente programma:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: i = 0;
5: while (i <= n) do
6: ris = ris + m;
7: i = i + 1;
8: end while

```

non è corretto siccome il predicato `ris == m*n` non vale una volta terminato.

2. Per induzione sul numero di iterazioni effettuate, dimostrare la correttezza parziale del problema PMIP che, dati due numeri naturali  $m$  e  $n$ , impone di calcolare il valore  $m^n$  come moltiplicazione di  $m$  iterata per  $n$  volte.

**Soluzione per PMIP.** Dato:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 1;
4: i = n;
5: while (i > 0) do
6: ris = ris * m;
7: i = i - 1;
8: end while

```

vogliamo dimostrare che, se il ciclo termina, allora la variabile `ris` contiene un valore pari a  $m^n$ .

La dimostrazione procede per induzione sul numero  $k$  di iterazioni, ovvero, sul numero di volte che l'intero corpo dell'iterazione:

```

ris = ris * m;
i = i - 1;

```

viene percorso.

Osserviamo che le variabili `ris` e `i` cambiano entrambe valore al termine di ogni ciclo. Ad esempio, dopo  $k$  cicli, per  $k$  opportuno, il valore di `ris` è diverso dal valore assunto dopo  $k+1$  cicli. Questo suggerisce di identificare come segue il valore di `ris` e `i` in funzione del numero  $0 \leq k$  di cicli percorsi:

- denotiamo con  $ris_k$  il valore della variabile `ris` al termine del  $k$ -esimo ciclo,
- denotiamo con  $i_k$  il valore della variabile `i` al termine del  $k$ -esimo ciclo.

### Proprietà 3

Siano  $m$  e  $n$  fissati. Per ogni  $0 \leq k \leq n$ , se la sequenza di istruzioni 6 e 7 è stata percorsa  $k$  volte, allora  $ris_k * (m^n) == m^n$  e  $i_k == (n-k)$ .

### Dim.

- Supponiamo  $k = 0$ . Il codice contiene le assegnazioni `ris=1` e `i=n` che sono eseguite prima di una qualsiasi iterazione. Quindi,  $ris_0==1$  e  $i_0==n$  e l'enunciato vale banalmente:  $ris_0 * (m^n) == m^n$  e  $i_0 == (n-0)$ .

- Dato  $0 < k < n$ , per ipotesi induttiva assumiamo  $\text{risk} * (\text{m}^{\wedge} i_k) == \text{m}^{\wedge} n$  e  $i_k == (n-k)$ . La prima equivalenza vale grazie ai seguenti passi:

$$\begin{aligned}
 \text{risk}_{k+1} * (\text{m}^{\wedge} i_{k+1}) &== (\text{risk}_k * \text{m}) * \text{m}^{\wedge} i_{k+1} && \text{(istruzione 4)} \\
 &== (\text{risk}_k * \text{m}) * \text{m}^{\wedge} (i_k - 1) && \text{(istruzione 5)} \\
 &== \text{risk}_k * (\text{m}^{\wedge} i_k) \\
 &== \text{m}^{\wedge} n && \text{(ipotesi induttiva)} .
 \end{aligned}$$

La seconda equivalenza vale grazie ai seguenti passi:

$$\begin{aligned}
 i_{k+1} &== i_k - 1 && \text{(istruzione 4)} \\
 &== (n-k) - 1 && \text{(ipotesi induttiva)} \\
 &== n - (k+1) .
 \end{aligned}$$

□

### Corollario 3

Siano  $m$  e  $n$  fissati. Dopo  $n$  cicli,  $\text{m}^{\wedge} n == \text{ris}$ .

**Dim.**

$$\begin{aligned}
 \text{m}^{\wedge} n &== \text{risk}_k * (\text{m}^{\wedge} i_k) && \text{(Proprietà 3 e } k == n\text{)} \\
 &== \text{risk}_n * (\text{m}^{\wedge} n_n) && \text{(Proprietà 3)} \\
 &== \text{risk}_n * (\text{m}^{\wedge} 0) \\
 &== \text{risk}_n * 1 \\
 &== \text{ris} .
 \end{aligned}$$

□

Prendendo spunto dalla precedente dimostrazione di correttezza parziale di PMIP, risolvere i seguenti esercizi:

(a) Scambiare le istruzioni 6 e 7 e dimostrare la correttezza parziale.

(b) Dimostrare la correttezza parziale del seguente algoritmo alternativo che risolve PMIP:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 1;
4: i = 0;
5: while (i < n) do
6: ris = ris * m;
7: i = i + 1;
8: end while

```

(c) Dimostrare che il seguente algoritmo, scritto per risolvere PMIP:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: i = 0;
5: while (i < n) do
6: ris = ris * m;
7: i = i + 1;
8: end while

```

non è corretto, siccome il predicato  $\text{ris} == \text{m}^{\wedge} n$  non vale una volta terminato.

3. Per induzione, dimostrare la correttezza parziale del problema DPPIP che, dati due numeri naturali  $m$  e  $n$ , impone di calcolare il valore  $m-n$  iterando un decremento unitario per  $n$  volte ad  $m$ .

**Soluzione per DPPIP.** Dato:

```

1: // m contiene un valore di N
2: // n contiene un valore di N
3: ris = 0;
4: if (m > n) then

```

```

5: ris = m;
6: i = n;
7: while (i > 0) do
8: ris = ris - 1;
9: i = i - 1;
10: end while
11: end if

```

vogliamo dimostrare che, se il ciclo termina, allora la variabile `ris` contiene un valore pari a  $m-n$ .

La dimostrazione procede per induzione sul numero  $k$  di iterazioni, ovvero, sul numero di volte che l'intero corpo dell'iterazione:

```

ris = ris - 1;
i = i - 1;

```

viene percorso.

Osserviamo che le variabili `ris` ed `i` cambiano entrambe valore al termine di ogni ciclo. Ad esempio, dopo  $k$  cicli, per  $k$  opportuno, il valore di `ris` è diverso dal valore assunto dopo  $k+1$  cicli. Questo suggerisce di identificare il valore di `ris` ed `i` in funzione del numero  $0 \leq k$  di cicli percorsi:

- denotiamo con  $ris_k$  il valore della variabile `ris` al termine del  $k$ -esimo ciclo,
- denotiamo con  $i_k$  il valore della variabile `i` al termine del  $k$ -esimo ciclo.

Lo scopo è dimostrare:

#### Proprietà 4

Siano  $m$  e  $n$  fissati. Per ogni  $0 \leq k \leq n$ , se la sequenza di istruzioni 8 e 9 è stata percorsa  $k$  volte, allora  $ris_{k+1} == m-n$  e  $i_k == (n-k)$ .

#### Dim.

- Supponiamo  $k = 0$ . Il codice contiene le assegnazioni `ris=m` e `i=n` che sono eseguite prima di una qualsiasi iterazione. Quindi,  $ris_0 == m$  e  $i_0 == n$  e l'enunciato vale banalmente:  $ris_0 - i_0 == m-n$  e  $i_0 == (n-0)$ .
- Dato  $0 < k < n$ , per ipotesi induttiva assumiamo  $ris_{k+1} == m-n$  e  $i_k == (n-k)$ . La prima equivalenza vale grazie ai seguenti passi:

$$\begin{aligned}
ris_{k+1} - i_{k+1} &= ris_k - i_{k+1} && \text{(istruzione 7)} \\
&= ris_k - 1 - (i_k - 1) && \text{(istruzione 8)} \\
&= ris_k - 1 - i_k + 1 \\
&= ris_k - i_k \\
&= m - n && \text{(ipotesi induttiva)} .
\end{aligned}$$

La seconda equivalenza vale grazie ai seguenti passi:

$$\begin{aligned}
i_{k+1} &= i_k - 1 && \text{(istruzione 8)} \\
&= (n-k) - 1 && \text{(ipotesi induttiva)} \\
&= n - (k+1) .
\end{aligned}$$

□

#### Corollario 4

Siano  $m$  e  $n$  fissati. Dopo  $n$  cicli,  $m-n == ris$ .

#### Dim.

$$\begin{aligned}
m - n &== ris_n - i_n && \text{(Proprietà 4 e } k == n) \\
&== ris_n - 0 && \text{(Proprietà 4)} \\
&== ris_n \\
&== ris .
\end{aligned}$$

□

#### 4.4.2 Correttezza parziale di QPNP

Richiamiamo il seguente algoritmo per il problema QPNP, già introdotto nella Sottosezione 4.2.2:

```

1: // x contiene un valore di N
2: n = 0;
3: ris = 0;
4: while (n < x) do
5: n = n + 1;
6: ris = ris + 2 * n - 1;
7: end while

```

Per induzione sul numero di iterazioni, dimostriamo  $\forall k. P(k)$  in cui (i), per definizione,  $P(k) \equiv n_k * n_k == ris_k$ , (ii)  $n_k$  e  $ris_k$  sono i valori di  $n$  e  $ris$ , rispettivamente, dopo aver percorso il ciclo per  $k$  volte, (iii) il legame tra  $n_k, n_{k+1}, ris_k$  e  $ris_{k+1}$  sia determinato dalle istruzioni 5 e 6 dell'algoritmo:

$$\begin{aligned} n_{k+1} &== n_k + 1 \\ ris_{k+1} &== ris_k + 2 * n_{k+1} - 1 . \end{aligned}$$

**Dim.** La dimostrazione procede per induzione su  $k$ .

$P(0)$  è vero perché  $n_0 * n_0 == 0 == ris$ .

Assumiamo valga  $P(k)$ . Allora:

|                                                    |               |
|----------------------------------------------------|---------------|
| $n_k * n_k == ris_k$                               | vero, implica |
| $n_k * n_k + 2 * n_k + 1 == ris_k + 2 * n_k + 1$   | vero, implica |
| $(n_k + 1)^2 == ris_k + (n_k + 1) + n_k$           | vero, implica |
| $(n_k + 1)^2 == ris_k + (n_k + 1) + (n_k + 1) - 1$ | vero, implica |
| $(n_k + 1)^2 == ris_k + 2 * (n_k + 1) - 1$         | vero, implica |
| $n_{k+1}^2 == ris_k + 2 * n_{k+1} - 1$             | vero, implica |
| $n_{k+1}^2 == ris_{k+1} .$                         | vero, implica |

□

#### 4.4.3 Correttezza parziale di QRDIP

Questo esempio ha, tra l'altro, lo scopo di presentare due stili diversi nell'esposizione di una dimostrazione di correttezza parziale, entrambi accettabili: il primo più narrativo, il secondo più formale.

##### Stile narrativo.

Consideriamo il problema di calcolare il quoziente  $q$  ed il resto  $r$  della divisione di due numeri interi  $X \geq 0$  e  $D > 0$ . L'algoritmo usuale consiste nel sottrarre ripetutamente  $D$  a  $X$ , aumentando ogni volta di 1 il valore di  $q$  che inizialmente ha valore 0. Schematicamente, l'algoritmo è il seguente:

1. fino a quando  $X \geq D$  esegui le seguenti azioni: sottrai  $D$  a  $X$ ; aumenta  $q$  di 1
2. quando  $X < D$ , pon  $r = X$ .

Un metodo Java che realizza questo algoritmo è il seguente:

```

public static void main (String[] args) {
 int X, D, q, r;
 X = 14;
 D = 3;
 q = 0;
 r = X;
 while (r >= D) {
 r = r - D;
 q = q + 1;
 }
}

```

Come si può dimostrare che il programma precedente è corretto? Prima di tutto, serve una specifica precisa del problema da risolvere: la condizione di ingresso del programma, cioè la proprietà che i dati in ingresso  $X$  e  $D$  devono soddisfare, è che  $X \geq 0$  e  $D > 0$  (la seconda proprietà serve ad evitare casi di divisione per 0). La condizione di uscita del programma, cioè la proprietà che i dati in uscita  $q$  ed  $r$  devono soddisfare, è che  $X = q * D + r$ , con  $r < D$ . Questa

proprietà dice proprio che  $q$  ed  $r$  sono, rispettivamente, il quoziente ed il resto della divisione intera di  $X$  per  $D$ . La correttezza del programma (qualche volta si parla di questa condizione come di CORRETTEZZA PARZIALE) asserisce che:

per ogni dato in ingresso che soddisfa la condizione di ingresso, se il programma termina, allora i dati in uscita soddisfano la condizione di uscita.

Una condizione più esigente di correttezza è quella che si chiama CORRETTEZZA TOTALE:

per ogni dato in ingresso che soddisfa la condizione di ingresso, *il programma termina* e i dati in uscita soddisfano la condizione di uscita.

Per stabilire che un programma soddisfa la specifica vi sono vari modi, ma la tecnica più conveniente consiste nel trovare quello che si chiama un INVARIANTE (di ciclo):

invariante (di un ciclo) è una proprietà che lega (tutte o alcune delle) le variabili coinvolte nel ciclo, e che è vera dopo un numero arbitrario di iterazioni del ciclo. In particolare, è vera all'ingresso nel ciclo (cioè dopo 0 iterazioni).

Ci sono molte proprietà invarianti del ciclo

```
1 while (r >= D) {
2 r = r - D;
3 q = q + 1;
}
```

nel programma precedente, per esempio la proprietà  $q \geq 0$ . Tra tutte le possibili proprietà ce ne sono alcune che sono più interessanti di altre. Consideriamo ora la proprietà:

$$X = q * D + r \quad (4.6)$$

che è molto simile alla condizione di uscita del programma. Che si tratti veramente di un invariante è qualcosa che deve ancora essere dimostrato, ma per il momento assumiamo che lo sia. Quando il ciclo termina (e prima o poi deve terminare, perché ad ogni iterazione a  $r$  viene sottratto il valore  $D$  che, per la condizione di ingresso, è un numero  $> 0$ , quindi prima o poi deve accadere che  $r < D$ ) abbiamo che  $X = q * D + r$  perché abbiamo assunto che questa proprietà sia invariante, ed inoltre si esce dal ciclo perché  $r < D$ . Ma allora è vera la proprietà  $X = q * D + r$ , con  $r < D$ , che è proprio la condizione di uscita del programma. L'uso dell'invariante ci permette quindi di dimostrare che il programma è (parzialmente) corretto. In questo caso abbiamo già implicitamente dimostrato che il programma è anche totalmente corretto, perché abbiamo già visto che il ciclo deve terminare. Resta da dimostrare che la proprietà (4.6) è proprio invariante. Questo si può fare per induzione sul numero di iterazioni del ciclo. Supponiamo che questo numero sia 0 (base dell'induzione) (ovviamente, la dimostrazione che (4.6) è invariante vale in generale, non solo per gli specifici valori di  $X$  e  $D$  che abbiamo scelto). Allora  $q = 0$  (perché  $q$  non viene incrementato) e  $r = X$ . Allora  $X = q * D + r$  perché questo si riduce a dire che  $X = 0 * D + X$ , che è ovviamente vero. Supponiamo che il ciclo sia stato eseguito  $n$  volte, e che la proprietà (4.6) sia vera (ipotesi induttiva); vogliamo dimostrare ora che resta vera anche dopo la  $(n+1)$ -esima iterazione. Durante questa iterazione vengono modificati i valori di  $q$  e di  $r$ , ottenendo valori

$$\begin{aligned} q' &= q + 1 \\ r' &= r - D \end{aligned}$$

dove  $q'$  ed  $r'$  sono i valori delle variabili  $q$  ed  $r$  dopo l'esecuzione delle istruzioni

```
r = r - D;
q = q + 1;
```

Allora calcoliamo:

$$\begin{aligned} q' * D + r' &= (q + 1) * D + (r - D) \\ &= q * D + D + r - D \\ &= q * D + r = X \end{aligned}$$

dove l'ultimo passaggio sfrutta l'ipotesi induttiva. Per induzione si conclude allora che la proprietà (4.6) è vera per qualsiasi numero di iterazioni del ciclo, quindi (4.6) è invariante.

### Stile formale

Per induzione sul numero di iterazioni effettuate, dimostrare la correttezza parziale del problema QRDIP che, dati due numeri naturali  $d$ , il divisore, ed  $s$ , il dividendo, impone di calcolare quoziente  $q$  e resto  $r$  della divisione intera di  $d$  per mezzo di  $s$ .

**Soluzione per QRDIP.** Dato il dividendo  $d$  ed il divisore  $s$  il predicato che esprime la relazione tra quoziente  $q$  e resto  $r$ , usando sottrazioni successive di  $s$  da  $d$ , avrà la forma seguente:

Dato:

```

1: // d contiene un valore di N
2: // s contiene un valore di N
3: r = d;
4: q = 0;
5: while (r >= s) do
6: r = r - s;
7: q = q + 1;
8: end while

```

vogliamo dimostrare  $\forall k. P(k)$  in cui (i), per definizione,  $P(k) \equiv d == q_k * s + r_k$ , (ii)  $q_k$  e  $r_k$  sono i valori di  $q$  e  $r$ , rispettivamente, dopo aver percorso il ciclo per  $k$  volte, (iii) il legame tra  $q_k, q_{k+1}, r_k$  e  $r_{k+1}$  sia determinato dalle istruzioni 4 e 5 dell'algoritmo:

$$\begin{aligned} r_{k+1} &== r_k - s \\ q_{k+1} &== q_k + 1 . \end{aligned}$$

**Dim.** La dimostrazione procede per induzione su  $k$ .

$P(0)$  è vero perché  $q == q_0 * s + d == d$ .

Assumiamo valga  $P(k)$ . Allora:

$$\begin{aligned} d &== q_{k+1} * s + r_{k+1} && \text{vero se} \\ d &== (q_k + 1) * s + (r_k - s) && \text{vero se} \\ d &== q_k * s + r_k - s + s && \text{vero se} \\ d &== q_k * s + r_k && \text{vero per ipotesi induttiva .} \end{aligned}$$

□

#### 4.4.4 Correttezza parziale per un algoritmo di calcolo del quadrato

Ancora adottando lo stile narrativo, vediamo un altro esempio della tecnica appena usata per dimostrare la correttezza del programma per la divisione intera, utilizzandola questa volta per sintetizzare un programma per calcolare il quadrato di un numero naturale  $N$ . La condizione di ingresso sarà dunque  $N \geq 0$ , mentre la condizione di uscita sarà  $Y = X * X$  e  $X = N$  dove  $Y$  è il dato in uscita ed  $X$  una variabile ausiliaria utilizzata come contatore. L'invariante appropriato in questo caso è la formula

$$Y = X * X. \quad (4.7)$$

Inizialmente avremo dunque  $X = 0$  e  $Y = 0$ : l'invariante è ovviamente vero in questo caso, e questo stabilisce la base della dimostrazione induttiva che la proprietà (4.7) è invariante.

```

1 class quadrato {
2 public static void main (String[] args) {
3 int N, X, Y;
4 N = ? ; // inizializzazione
5 X = 0;
6 Y = 0;
7 while (X < N) {
8 Y = Y + 2 * X + 1;
9 X = X + 1;
10 }
11 System.out.println ("Quadrato = " + Y);
12 }
}

```

Per quanto riguarda il passo induttivo, l'ipotesi induttiva è

$$Y = X * X \text{ dopo l'}n\text{-esima iterazione;}$$

bisogna dimostrare che (4.7) resta vera dopo la  $(n + 1)$ -esima iterazione. Se  $Y'$  è il valore di  $Y$  dopo l'esecuzione dell'istruzione  $Y = Y + 2 * X + 1$ , mentre  $X'$  è il valore di  $X$  dopo l'esecuzione dell'istruzione  $X = X + 1$ , possiamo calcolare

$$\begin{aligned} Y' &= Y + 2 * X + 1 \\ &= (X * X) + 2 * X + 1 && \text{(per ipotesi induttiva)} \\ &= (X + 1) * (X + 1) \\ &= X' * X' \end{aligned}$$

da cui si conclude che (4.7) è proprio invariante. Poiché il valore di  $N - X$  decresce strettamente ad ogni iterazione, il ciclo deve terminare (perché non ci può essere una sequenza infinita di numeri naturali  $k_0 > k_1 > k_2 > \dots$ ) all'uscita dal ciclo avremo  $X = N$  (perché la condizione del while è diventata falsa e sappiamo, per come è fatto il programma, che  $X \leq N$ ) quindi, per l'invariante,  $Y = N * N$ . Questo mostra che la condizione di uscita è soddisfatta dal dato in uscita  $Y$ , perciò il programma è corretto.

**Esercizio 22** 1. Partendo dalla dimostrazione precedente, verificare per induzione sul numero di iterazioni se il predicato:

$$x^2 = x + 2 * \sum_{i=1}^x (x - i)$$

sia un invariante di ciclo per [QuadratoConRaddoppiEtc.java](#).

2. (*Difficile.*) Dimostrare la correttezza parziale de [QuadratoDiStefanoMerlo.java](#) individuando un predicato  $P(k)$  che, usando un ragionamento per induzione sul numero di cicli percorsi, sia l'invariante.

## 4.5 Correttezza parziale con prediciati implicativi

Partiamo ricordando la tavola di verità di implicazione e quantificazione universale.

L'implicazione è sempre vera, tranne quando la premessa è vera e la conclusione falsa:

| X     | Y     | $X \Rightarrow Y$ |
|-------|-------|-------------------|
| false | false | true              |
| false | true  | true              |
| true  | false | false             |
| true  | true  | true              |

La quantificazione universale è vera se la congiunzione di tutte le istanze del predicato su cui si quantifica è vera.

*Particolare interesse per noi ha il modo in cui si verifichi la validità di un predicato che quantifichi universalmente su una implicazione.*

**Esempio 7 (Validità di una quantificazione universale su un'implicazione)**  $\forall k \in \mathbb{N}. 0 \leq k < 3 \Rightarrow 3 - k \geq 0$  è vero, perché equivalente a quanto segue:

$$\begin{array}{lll} 0 \leq 0 < 3 \Rightarrow 3 - 0 \geq 0 \wedge & \text{true} \Rightarrow \text{true} \wedge & \text{true} \wedge \\ 0 \leq 1 < 3 \Rightarrow 3 - 1 \geq 0 \wedge & \text{true} \Rightarrow \text{true} \wedge & \text{true} \wedge \\ 0 \leq 2 < 3 \Rightarrow 3 - 2 \geq 0 \wedge & \text{true} \Rightarrow \text{true} \wedge & \text{true} \wedge \\ 0 \leq 3 < 3 \Rightarrow 3 - 3 \geq 0 \wedge & \text{false} \Rightarrow \text{true} \wedge & \text{true} \wedge \Leftrightarrow \text{true} \wedge \Leftrightarrow \text{true} . \\ 0 \leq 4 < 3 \Rightarrow 3 - 4 \geq 0 \wedge & \text{false} \Rightarrow \text{false} \wedge & \text{true} \wedge \\ 0 \leq 5 < 3 \Rightarrow 3 - 5 \geq 0 \wedge & \text{false} \Rightarrow \text{false} \wedge & \text{true} \wedge \\ \vdots & \vdots & \vdots \end{array}$$

■

La congiunzione più a sinistra nell'Esempio 7 è ottenuta eliminando la quantificazione universale da  $\forall k \in \mathbb{N}. 0 \leq k < 3 \Rightarrow 3 - k \geq 0$ . L'eliminazione consiste nel sostituire a  $k$  in  $0 \leq k < 3 \Rightarrow 3 - k \geq 0$  ogni possibile valore che  $k$  stesso possa assumere. Le congiunzioni centrali e seguenti risultano dalla valutazione dei prediciati che compongono ciascuna implicazione. L'aspetto interessante è che la congiunzione infinita è vera non perché in essa compaiano solo implicazioni con premessa e conclusione vere. Al contrario, essa contiene sia implicazioni  $\text{false} \Rightarrow \text{true}$ , sia implicazioni  $\text{false} \Rightarrow \text{false}$ .

*La possibilità di scrivere prediciati in cui l'implicazione sia vera grazie al fatto che la premessa è falsa gioca un ruolo fondamentale nelle dimostrazioni di correttezza parziale che illustreremo in questa sezione.*

**Esercizio 23** Seguendo l'Esempio 7 determinare il valore di verità dei seguenti predicati:

- $Q \equiv \forall k \in \mathbb{N}. 0 \leq k < 2 \Rightarrow 2 - k > 0$ .
- $R \equiv \forall k \in \mathbb{N}. 0 \leq k < 2 \Rightarrow 2 - k \leq 0$ .
- $S \equiv \forall k \in \mathbb{N}. 0 \leq k \leq 1 \Rightarrow 1 - k \geq 1$ .
- $T \equiv \forall k \in \mathbb{N}. k \text{ è pari} \Rightarrow k + 1 \text{ è dispari}$ . ■

## 4.6 Correttezza parziale per SNNP

Per definizione, il problema SNNP consiste nel voler stampare l'intera sequenza di numeri naturali tra 0 ed un valore prefissato, primo estremo incluso e secondo escluso.

Se il valore prefissato è 2, un algoritmo che risolva SNNP dovrà produrre la sequenza 01. Proponiamo l'Algoritmo 12 seguente come soluzione.

---

### Algoritmo 12 per SNNP

---

```

1: // n contiene un valore di N
2: i = 0;
3: while (i < n) do
4: stampa i;
5: i = i + 1;
6: end while

```

---

*Il nostro problema è dimostrare che l'Algoritmo 12 sia parzialmente corretto.*

A tal fine, occorre sintetizzare un predicato  $P$  che sia vero in ogni configurazione del programma. Ovvero,  $P$  deve valere in tutti i casi seguenti:

1. l'Algoritmo 12 non ha ancora stampato nulla. Questo corrisponde alla situazione in cui abbiamo interpretato solo l'assegnazione  $i = 0$ ; e stiamo per, ma non lo abbiamo ancora fatto, interpretare per la prima volta il comando `stampa i`;
2. l'Algoritmo 12 ha stampato il valore 0, ha interpretato  $i = i + 1$ , ma non ha ancora valutato per la seconda volta l'espressione  $i < n$  per decidere se rieseguire il corpo dell'iterazione;
3. e così via fino a quando  $i$  non supera il valore in  $n$ .

Definiamo  $P$  e argomentiamo sul perché esso possa esser un candidato per dimostrare la correttezza parziale dell'Algoritmo 12. Sia:

$$P(i) \equiv \forall k. 0 \leq k < i \Rightarrow \text{il valore } k \text{ è già stato stampato}. \quad (4.8)$$

Sappiamo che il valore di verità del predicato  $P(i)$  si determina sviluppandolo in una congiunzione infinita e valutando ogni implicazione in ogni congiunto:

$$\begin{aligned} 0 \leq 0 < i &\Rightarrow 0 \text{ è già stato stampato} \wedge \\ 0 \leq 1 < i &\Rightarrow 1 \text{ è già stato stampato} \wedge \\ &\vdots \\ 0 \leq i-1 < i &\Rightarrow i-1 \text{ è già stato stampato} \wedge \\ 0 \leq i < i &\Rightarrow i \text{ è già stato stampato} \wedge \\ 0 \leq i+1 < i &\Rightarrow i+1 \text{ è già stato stampato} \wedge \\ &\vdots \end{aligned}$$

Dovrebbe essere evidente che il valore di  $P(i)$  può dipendere dal valore di  $i$ . Proviamo alcuni casi:

- Sia  $i$  pari a 0. Allora:

$$\begin{array}{lll} 0 \leq 0 < 0 \Rightarrow 0 \text{ è già stato stampato} \wedge & \text{false} \Rightarrow 0 \text{ è già stato stampato} \wedge & \text{true} \wedge \\ 0 \leq 1 < 0 \Rightarrow 1 \text{ è già stato stampato} \wedge & \text{false} \Rightarrow 1 \text{ è già stato stampato} \wedge & \text{true} \wedge \\ 0 \leq 2 < 0 \Rightarrow 2 \text{ è già stato stampato} \wedge \Leftrightarrow & \text{false} \Rightarrow 2 \text{ è già stato stampato} \wedge \Leftrightarrow & \text{true} \wedge \Leftrightarrow \text{true}. \\ \vdots & \vdots & \vdots \end{array}$$

In questo caso, ogni implicazione è vera grazie alla tavola di verità dell'implicazione, anche se *nessuno* dei valori 0, 1, 2, etc. sia stato effettivamente stampato. Questo è utile perché, ad esempio, interpretando la linea 3 per la prima volta, nulla è stato stampato, ma il predicato deve essere vero.

- Sia  $i$  pari a 1. Allora:

$$\begin{array}{lll}
 0 \leq 0 < 1 \Rightarrow 0 \text{ è già stato stampato} \wedge & \text{true} \Rightarrow 0 \text{ è già stato stampato} \wedge & \text{true} \wedge \\
 0 \leq 1 < 1 \Rightarrow 1 \text{ è già stato stampato} \wedge & \text{false} \Rightarrow 1 \text{ è già stato stampato} \wedge & \text{true} \wedge \\
 0 \leq 2 < 1 \Rightarrow 2 \text{ è già stato stampato} \wedge \Leftrightarrow & \text{false} \Rightarrow 2 \text{ è già stato stampato} \wedge \Leftrightarrow & \text{true} \wedge \Leftrightarrow \text{true} . \\
 \vdots & \vdots & \vdots
 \end{array}$$

In questo caso la prima implicazione è vera solo se il valore 0 è stato effettivamente stampato. Tutte le implicazioni seguenti, invece, valgono semplicemente perché la premessa è falsa. Quindi,  $P(1)$  è vera perché è stato stampato il valore 0 e descrive la configurazione subito dopo aver interpretato la riga 5.

- Potremmo continuare per ogni valore di  $i$ , constatando che  $P(i)$  è vero grazie al fatto che un numero crescente di valori sono stati effettivamente stampati. Il modo corretto di procedere sarebbe dimostrare  $\forall i.P(i)$  per induzione, ovvero occorrerebbe dimostrare:

$$(P(0) \wedge (\forall i.P(i) \Rightarrow P(i+1))) \Rightarrow (\forall i.P(i)) .$$

Siccome  $P(i)$  sembra descrivere convincentemente un processo di stampa incrementale di un certo numero di valori, constatiamo che esso si adatta a descrivere le configurazioni dell'Algoritmo 12. Riprendiamo quest'ultimo qui sotto, includendo la dimostrazione di correttezza parziale “per riscrittura”:

```

1: // n contiene un valore di N
2: i = 0;
3: // P(0)
4: while (i < n) do
5: // P(i)
6: stampa i;
7: /* P(i) && i è stato stampato.
8: Quindi P(i+1) perché ora sullo schermo
9: ci sono i valori 0, 1, ..., i-1, i
10: */
11: i = i + 1;
12: // P(i) perché i è un nome per i+1.
13: end while
14: /* P(i) && i == n implica P(n).
15: Quindi, sullo schermo ci sono i valori 0, 1, ..., n-1.
16: */

```

#### Esercizio 24 (Correttezza parziale con prediciati implicativi)

1. Sia dato il seguente algoritmo:

```

1: // n contiene un valore di N
2: i = 0;
3: while (i < n) do
4: leggi v;
5: stampa v;
6: i = i + 1;
7: end while

```

Esso legge  $n$  valori recuperati attraverso un qualche mezzo, ad esempio la tastiera, e, man mano che la lettura procede, stampa il valore appena letto. Verificare che la dimostrazione di correttezza parziale “per riscrittura” possa essere portata a termine usando il seguente predicato:

$$P(i) \equiv \forall k. 0 \leq k < i \Rightarrow k\text{-esimo valore } v \text{ è stato letto e scritto} .$$

2. Sia dato il seguente algoritmo:

```

1: // n contiene un valore di N
2: i = 0;
3: r = 0;
4: while (i < n) do
5: leggi v;
6: r = r + v;
7: i = i + 1;
8: end while

```

Esso legge  $n$  valori recuperati attraverso un qualche mezzo, ad esempio la tastiera, e, man mano, ne accumula la somma in  $r$ . Verificare che la dimostrazione di correttezza parziale “per riscrittura” possa essere portata a termine usando il seguente predicato:

$$P(i) \equiv r == \sum_{k=0}^{i-1} k\text{-esimo valore v letto} .$$

3. Dimostrare la correttezza parziale dei due seguenti algoritmi che forniscono una possibile soluzione a PDMP, problema che, dato  $m \in \mathbb{N}$ , restituisce vero se  $n$  è pari e falso altrimenti. Il primo algoritmo è:

```

1: // n contiene un valore di N
2: i = n;
3: while (i > 1) do
4: i = i - 2;
5: end while
6: r = i == 0;
```

Il secondo algoritmo è:

```

1: // n contiene un valore di N
2: i = 0;
3: pari = true;
4: while (i < n) do
5: i = i + 1;
6: pari = !pari;
7: end while
```

`PDMPDecrementiDue.java` è una classe Java con la dimostrazione di correttezza parziale per il primo algoritmo mentre `PDMPFlipFlop.java` quella per il secondo. ■

**Esercizio 25 (Correttezza parziale con predicati implicativi e lettura da tastiera)** Per svolgere agevolmente l'esercizio anche a livello di programmazione, può servire l'abitudine all'uso della classe `SIn.java`, documentata in [SIn Javadoc](#).

1. Scrivere in algoritmo che, fissato un valore  $n \geq 1$ , legge  $n$  interi e stampa il massimo tra essi incontrato. Quindi, dimostrarne la correttezza parziale.

Una soluzione espressa nel linguaggio di riferimento è in [MassimoTraInteri.java](#).

2. Scrivere un algoritmo che legga iterativamente numeri naturali da tastiera. La lettura termina all'inserimento del valore 0. Man mano che i numeri vengono letti, occorre contare sia le occorrenze di numeri pari, sia quelle di numeri dispari. Al termine della lettura occorre essere in grado di sapere, ad esempio stampandoli, quanti dispari e quanti pari sono stati incontrati, escludendo lo 0 finale. Quindi, dimostrarne la correttezza parziale.

Una soluzione espressa nel linguaggio di riferimento è in [ContaPariDispari.java](#). ■

#### 4.6.1 Iterazioni annidate

Fissati due numeri naturali  $m$  ed  $n$ , il seguente algoritmo stampa una matrice di simboli  $*$  composta da  $m$  righe ed  $n$  colonne.

```

1: // m contiene un valore in N
2: // n contiene un valore in N
3: i = 0;
4: j = 0;
5: while (i < m) do
6: while (j < n) do
7: stampa *;
8: j = j + 1;
9: end while
10: i = i + 1;
11: j = 0;
12: end while
```

L'algoritmo è caratterizzato da un paio di iterazioni annidate. L'iterazione più interna ha l'obiettivo di stampare una riga completa con  $n$  asterischi. Tale riga è sotto un certo numero di righe già completamente stampate. Il ciclo più esterno ha il compito di cambiare riga di stampa non appena il ciclo più interno ha terminato una riga.

Genericamente, “nel mezzo” della sua interpretazione, l'algoritmo ha generato la seguente illustrazione:



Sono state stampate *completamente*  $i-1$  righe, ciascuna con  $n$  asterischi. La riga più in basso di indice  $i$  è in via di stampa e conterrà  $j$  asterischi.

La situazione è descrivibile con il seguente predicato:

```

se k == 0, allora la k-esima riga è già completamente stampata e
se k == 1, allora la k-esima riga è già completamente stampata e ...
... e se k == i-2, allora la k-esima riga è già completamente stampata e
 se k == i-1, allora la k-esima riga è già completamente stampata e
la i-esima riga contiene j occorrenze di * .

```

È possibile comprimere la formalizzazione del predicato precedente, sfruttando un quantificazione universale:

```

 $P(i, j) \equiv$ per ogni k,
 se $0 \leq k < i$, allora la k-esima riga è già completamente stampata
 e la i-esima riga contiene j occorrenze di * .

```

Per comprende bene di cosa predichi il predicato  $P(i, j)$  fissiamo alcune combinazioni di valori per  $i$  e  $j$ :

- $P(0, 0)$  è vero e descrive una situazione in cui non sia stato stampato alcun  $*$  in alcuna riga perché:

```

per ogni k,
se $0 \leq k < 0$, allora la k-esima riga è già completamente stampata
e la 0-esima riga contiene 0 occorrenze di *
equivale a: per ogni k,
 se false, allora la k-esima riga è già completamente stampata
 e la 0-esima riga contiene 0 occorrenze di *
equivale a: per ogni k, true e true
che equivale a: true .

```

- $P(i, 0)$  è vero e descrive la situazione in cui tutte le righe con indice comprese nell'intervallo  $[0, i)$  sono state completate. In particolare,  $P(m, 0)$  indica che sono state completate esattamente  $m$  righe.
- Per valori intermedi di  $i$  e  $j$  il predicato è vero e descrive una situazione generica come quella rappresentata nell'illustrazione (4.9).

Il predicato  $P(i, j)$  si presta ad essere l'invariante del seguente algoritmo che, passo dopo passo, incrementa sia la quantità di asterischi stampati nella riga ancora da completare, sia l'insieme delle righe completate:

```

1: // m contiene un qualche valore in N
2: // n contiene un qualche valore in N
3: i = 0;
4: j = 0;
5: // P(0, 0)
6: while (i < m) do
7: // P(i, j)
8: while (j < n) do
9: // P(i, j)
10: stampa *;
11: // P(i, j + 1) perché è appena stato stampato * di posizione j nella riga i-esima
12: j = j + 1;
13: // P(i, j) perché j rappresenta l'espressione j+1
14: end while
15: /* j == n perché il ciclo più interno è terminato.
16: Quindi è stata completamente stampata la i-esima riga. Allora P(i + 1, j) è vero
17: */
18: i = i + 1; cambia riga;

```

```

19: // P(i,0) perché i rappresenta l'espressione i + 1 e la riga con tale indice non ha
*
20: j = 0;
21: // P(i,j) perché ora j è un nome per 0
22: end while
23: // i == m && j == 0 && P(m,0).

```

**Esercizio 26 (Correttezza parziale con implicazioni ed iterazioni annidate)**

- Il seguente algoritmo:

```

1: i = 1;
2: j = 1;
3: while (i<11) do
4: while (j<11) do
5: stampa i*j;
6: j = j + 1;
7: end while
8: i = i + 1;
9: cambia riga;
10: j = 1;
11: end while

```

stampa una matrice che costituisce la tavola pitagorica dei primi dieci numeri interi.

Verificare che il seguente predicato possa essere usato per dimostrarne la correttezza parziale:

$$\begin{aligned}
 P(i, j) \equiv & \text{ per ogni } k, \\
 & \text{se } 1 \leq k < i, \text{ allora } k\text{-esima tabellina completamente stampata} \\
 & \text{e per ogni } k', \text{ se } 1 \leq k' < j, \text{ allora il } i*k' \text{ è stato stampato} .
 \end{aligned}$$

[TavolaPitagoricaWhile.java](#) è un programma nel linguaggio di riferimento, commentato con la dimostrazione di correttezza parziale “per riscrittura”.

- Il seguente algoritmo:

```

1: // m contiene un qualche valore in N
2: // n contiene un qualche valore in N
3: i = 0;
4: j = 0;
5: while (i < m && i < n) do
6: while (j < i) do
7: stampa *;
8: j = j + 1;
9: end while
10: i = i + 1;
11: cambia riga;
12: j = 0;
13: end while

```

fissa due numeri naturali m ed n e stampa un triangolo, usando il carattere \*. Il triangolo è equilatero. La lunghezza del lato è pari al minimo tra m ed n.

Verificare che il seguente predicato possa essere usato per dimostrarne la correttezza parziale:

$$\begin{aligned}
 P(i, j, m) \equiv & \text{ per ogni } k, \text{ se } 0 \leq k < i, \text{ allora } k\text{-esima riga ha } k \text{ occorrenze di *} \\
 & \text{e } i\text{-esima riga ha } j \text{ occorrenze di *} .
 \end{aligned}$$

[TriangoloDiStelle.java](#) è un programma nel linguaggio di riferimento, commentato con la dimostrazione di correttezza parziale “per riscrittura”.

- Il seguente algoritmo:

```

1: // m contiene un qualche valore in N
2: // n contiene un qualche valore in N
3: i = 0;
4: j = 0;
5: while (i < m) do
6: while (j < i && j < n) do

```

```

7: stampa *;
8: j = j + 1;
9: end while
10: i = i + 1;
11: cambia riga;
12: j = 0;
13: end while

```

fissa due numeri naturali  $m$  ed  $n$  e, usando il simbolo  $*$ , stampa un triangolo equilatero di base  $m$ , se  $m \leq n$ . Altrimenti, stampa un trapezio con base maggiore di  $m$  righe e base minore di  $m-n$  righe.

Verificare che il seguente predicato possa essere usato per dimostrarne la correttezza parziale:

```

se i <= n, allora
 per ogni k,
 se 0 <= k < i, allora
 k-esima riga ha k occorrenze di *
 e i-esima riga ha j occorrenze di *
 e se n < i, allora
 per ogni k,
 se 0 <= k < n, allora
 k-esima riga ha k occorrenze di *
 e se n <= k < i, allora k-esima riga ha n occorrenze di *
 e i-esima riga ha j occorrenze di * .

```

`TrapezioDiStelle.java` è un programma nel linguaggio di riferimento, commentato con la dimostrazione di correttezza parziale “per riscrittura”. ■

#### 4.6.2 Iterazioni annidate e correttezza senza prediciati implicativi

La sottosezione precedente sembra indicare che la correttezza di un algoritmo con iterazioni annidate sia possibile essenzialmente solo sfruttando prediciati che contengano implicazioni.

Per completezza vediamo la correttezza di un algoritmo con iterazioni annidate che non si basi su un predicato contenente un’implicazione. Riprendiamo il problema MSDIP<sup>”</sup> il cui scopo era moltiplicare due numeri naturali, il moltiplicando  $m$  ed il moltiplicatore  $M$ , utilizzando solo incrementi e decrementi di una unità. Sfruttando l’esperienza maturata sinora nell’individuare la configurazione “nel mezzo”, possiamo descrivere quest’ultima come segue:

$$\underbrace{(1 + \dots + 1)}_N + \dots + \underbrace{(1 + \dots + 1)}_N + \underbrace{(1 + \dots + 1 + 1 + \dots + 1)}_{m-n} + \underbrace{(1 + \dots + 1)}_{M-(N+1)} + \dots + \underbrace{(1 + \dots + 1)}_M$$

nella quale  $r$  indica il risultato. Si può descrivere la configurazione col predicato:

$$m * M = r + (m - n) + m * (M - (N + 1)) \wedge r = m * N + n \quad (4.10)$$

che si può leggere come segue:

- nel risultato  $r$  ci sono  $N$  blocchi già completi, cioè con  $m$  uni, ed uno parziale con solo  $n < m$  uni,
- Il resto delle unità non in  $r$ , ovvero  $m * (M - (N + 1))$  sono quelle che mancano per ottenere  $m * M$ .

`MSIDPCicliAnnidati` è la classe Java che implementa un algoritmo corrispondente alle regole di riscrittura elencate nei commenti e la dimostrazione di correttezza parziale. Possiamo osservare che, al contrario di altre versioni di MSDIP<sup>”</sup>, usiamo solo incrementi di una unità.

#### 4.6.3 Approfondimenti facoltativi

I seguenti riferimenti sono letture integrative per i più curiosi a proposito di correttezza parziale degli algoritmi:

- [Wir73, Capitolo 5] che introduce la notazione a *flow chart* dei programmi. Il vantaggio della notazione può essere il rendere più leggibile la natura iterativa del costrutto while-do.
- Hoare logic – wikipedia.
- L’articolo `Formula Slicing: Inductive Invariants from Preconditions` è un recente articolo scientifico centrato sulla sintesi automatica di proprietà invarianti.

# Capitolo 5

## Programmazione ricorsiva di base

Se dovessimo riassumere in maniera essenziale il principio guida alla base della programmazione iterativa potremmo affermare che essa costruisce un risultato, accumulando, passo dopo passo, sue approssimazioni sempre migliori.

Il principio guida intuitivo alla base della programmazione ricorsiva è una particolare visione dello schema *divide et impera*. Essa consiste nello scomporre un problema dato in modo che esso, il problema, si ripresenti, ma in situazioni più semplici da risolvere. Una volta ottenute le soluzioni alle versioni più semplici, le si compongono opportunamente per ottenere la soluzione alla versione meno banale del problema dato.

Immaginiamo di volere definire una funzione  $f : N \rightarrow A$ , dove  $N$  è l'insieme dei numeri naturali ed  $A$  un insieme qualsiasi. Si può allora utilizzare il seguente schema di ricorsione:

$$f(0) = a \tag{5.1}$$

$$f(n) = E(f(n - 1)) \quad (n > 0) \tag{5.2}$$

dove  $a$  è un elemento di  $A$  e la notazione  $E(f(n - 1))$  indica che l'espressione  $E$  può utilizzare al suo interno il valore  $f(n - 1)$ . Una giustificazione intuitiva di questo schema si può ottenere considerando la struttura dei numeri naturali: la funzione  $f$  è definita per 0 perché la prima clausola dello schema ne fornisce il valore  $a$ ; supponiamo invece che  $n$  sia un numero positivo. Si può immaginare di avere già calcolato il valore di  $f(n - 1)$  (la funzione  $f$  viene calcolata “dal basso”, partendo dall'argomento 0) per poi calcolare  $E(f(n - 1))$  che dà il valore di  $f(n)$ .

Il riferimento per la programmazione ricorsiva è [SM14, Capitolo 7]. In particolare, la Sezione 7.2.1 in [SM14, Capitolo 7] parla del ragionamento induttivo che sta alla base della progettazione di programmi ricorsivi.

### 5.1 Definizioni e computazioni ricorsive

Forniamo esempi di base di algoritmi definiti ricorsivamente.

#### 5.1.1 Fattoriale

Focalizziamo la nostra attenzione sul calcolo del fattoriale  $n!$  di  $n$ . Un modo per descrivere il valore di  $n!$  è elencare ogni fattore moltiplicativo che compone la sua definizione:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times (n - (n - 2)) \times (n - (n - 1)) . \tag{5.3}$$

Se ci chiediamo quanto valga  $(n - 1)!$ , ponendo  $n - 1 = m$ , possiamo procedere in maniera analoga, con  $m$  al posto di  $n$ :

$$m! = m \times (m - 1) \times (m - 2) \times \cdots \times (m - (m - 2)) \times (m - (m - 1)) . \tag{5.4}$$

Sostituendo  $n - 1$  ad  $m$  in 5.4, lo sviluppo diventa:

$$(n - 1)! = (n - 1) \times (n - 2) \times \cdots \times (n - (n - 2)) \times (n - (n - 1)) . \tag{5.5}$$

Osserviamo che 5.5 compare, tale e quale, in 5.3 moltiplicato per  $n$ . A questo punto, la sostituzione di 5.5 in 5.3 è cruciale per comprendere il nocciolo della questione:

$$n! = n \times (n - 1)! . \tag{5.6}$$

In 5.6 il problema di calcolare il valore  $n!$  è spezzato in due sotto-problemi.

Il primo consiste nel calcolare il fattoriale di  $n - 1$ . Ovvero, stiamo riproponendo il problema iniziale, ma in una veste più semplice, perché relativo ad un valore strettamente più piccolo di quello iniziale.

Supponiamo, quindi, di essere in grado di calcolare  $(n - 1)!$ .

Allora, conoscendo sia  $(n - 1)!$ , sia  $n$ , sappiamo anche risolvere il secondo problema che consiste nel calcolare  $n \times (n - 1)!$ , ovvero  $n!$ .

Si dice che abbiamo *ridotto* il calcolo di  $n!$  a quello di  $(n - 1)!$ .

Si potrebbe avere l'impressione che il meccanismo che stiamo impostando non sia efficace perché "senza fine".

Sperimentalmente, è possibile rendersi conto che questo non sia il caso.

Supponiamo, ad esempio, di voler calcolare concretamente il valore  $3!$ , come indicato dalla scomposizione vista:

- Per definizione,  $3!$  richiede di calcolare  $2!$ . Una volta ottenuto  $2!$ , lo moltiplichiamo per  $3$  ed otteniamo  $3!$ . Calcoliamo, quindi,  $2!$ .
- Per definizione,  $2!$  richiede di calcolare  $1!$ . Una volta ottenuto  $1!$ , lo moltiplichiamo per  $2$  ed otteniamo  $2!$ . Calcoliamo, quindi,  $1!$ .
- Per definizione,  $1!$  richiede di calcolare  $0!$ . Una volta ottenuto  $0!$ , lo moltiplichiamo per  $1$  ed otteniamo  $1!$ . Calcoliamo, quindi,  $0!$ .
- Il valore di  $0!$  deve essere primitivo. Se non lo fosse, dovremmo applicare la definizione  $0! = 0 \times (0 - 1) = 0 \times (-1) = 0$ , risultato che azzererebbe qualsiasi valore moltiplicato per esso. Quindi,  $0!$  deve essere  $1$ , valore neutro per la moltiplicazione.

Stabilito che  $0! = 1$ , possiamo, passo dopo passo, eseguire tutte le moltiplicazioni rimaste in sospeso nei passi precedenti.

- Il valore  $1!$  è  $1 \times 0! = 1 \times 1 = 1$ .
- Il valore  $2!$  è  $2 \times 1! = 2 \times 1 = 2$ .
- Il valore  $3!$  è  $3 \times 2! = 3 \times 2 = 6$  ed il calcolo è concluso.

Il processo di calcolo descritto è riassumibile come segue:

$$\begin{aligned} 3! &= 3 \times 2! \\ &= 3 \times (2 \times 1!) \\ &= 3 \times (2 \times (1 \times 0!)) \\ &= 3 \times (2 \times (1 \times 1)) \\ &= 3 \times (2 \times 1) \\ &= 3 \times 2 \\ &= 6 . \end{aligned}$$

Siamo nella condizione di poter riformulare  $n!$  in *forma ricorsiva*, come segue:

$$x! = \begin{cases} 1 & \text{se } x = 0 \\ x \times (x - 1)! & \text{se } x > 0 \end{cases} . \quad (5.7)$$

La definizione è "ricorsiva" perché la funzione che stiamo definendo, compare nella definizione stessa, ma applicata a valori decrescenti dell'argomento. Più formalmente, possiamo dire che (5.7) è una definizione ricorsiva perché essa assume la forma di riferimento (5.1) che riportiamo qui per comodità:

$$\begin{aligned} f(0) &= a \\ f(n) &= E(f(n - 1)) \end{aligned} \quad (n > 0) .$$

Qui sopra,  $E(y)$  sta per la funzione:

$$x \times y$$

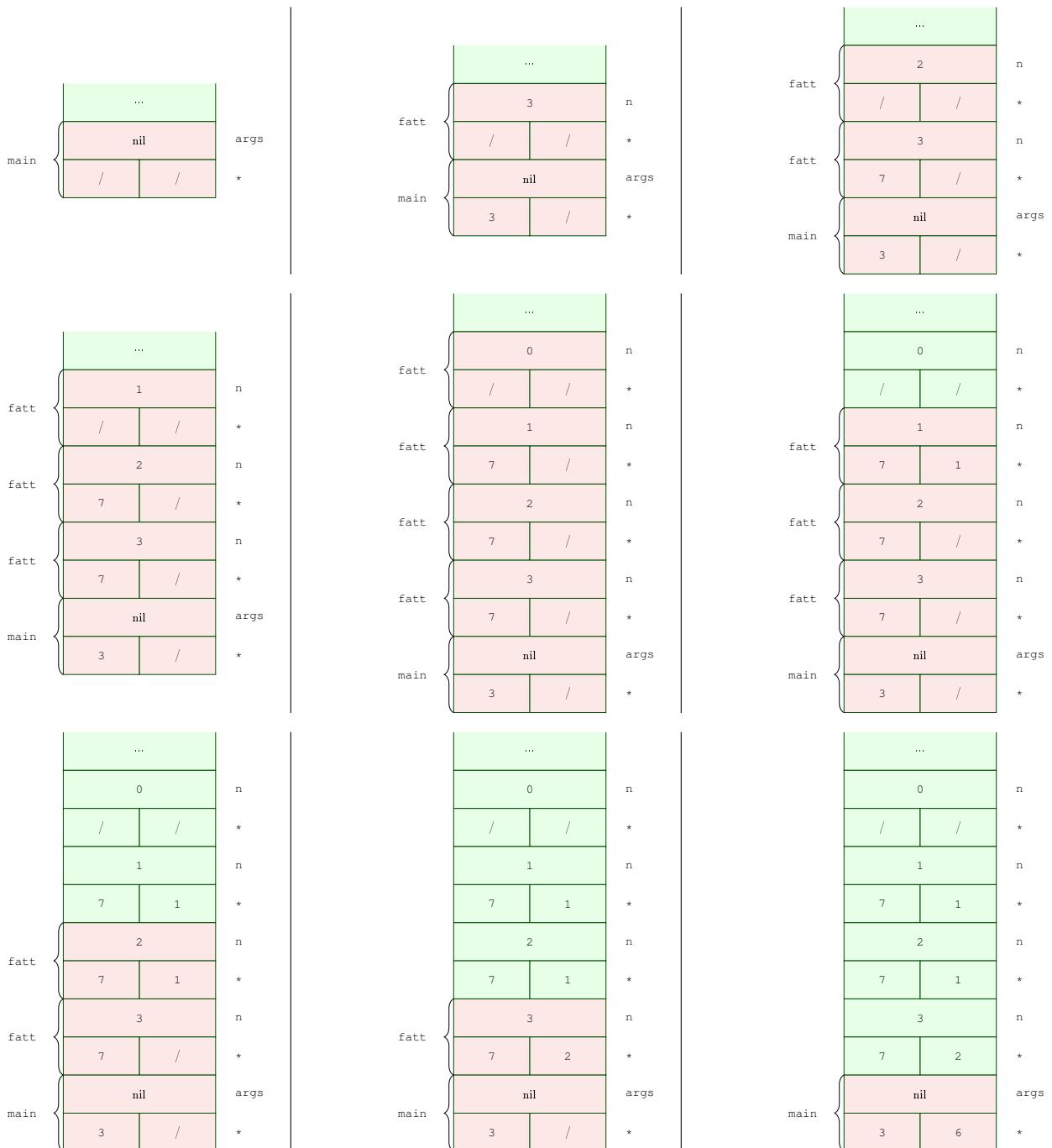
mentre  $f(n - 1)$  coincide con  $(n - 1)!$ .

Inoltre, la definizione (5.7) si presta alla traduzione immediata nel programma `FattorialeRec.java` del quale, per comodità, riportiamo il sorgente in forma essenziale:

```

1 public class FattorialeRec {
2
3 public static int fatt (int n) {
4 if (n == 0)
5 return 1;
6 else
7 return n * fatt(n - 1);
8 }
9
10 // Versione che sfrutta l'espressione condizionale disponibile in Java
11 public static int fattAlt (int n) {
12 return (n == 0 || n == 1) ? 1 : n * fattAlt(n - 1);
13 }
14 }
```

La definizione del metodo `fatt`, ad esempio, contiene un richiamo a se stessa. Come in precedenza, si può avere l'impressione di una definizione mal fondata, ovvero che permetta un richiami infinito della definizione stessa. Un primo mezzo per convincerci del contrario è simulare una interpretazione. Riportiamo una sequenza rilevante di "istantanee" del *frame stack* conseguente alla chiamata `fatt(3)`; del main:



Nell'ultima configurazione è attivo solo il *frame* del *main* che contiene il risultato ottenuto dall'ultimo *frame* di *fatt* disallocato.

**Esercizio 27 (Stack della chiamata **fattAlt(3)**.)** Disegnare una sequenza rilevante delle configurazioni della memoria durante l'interpretazione della chiamata *fattAlt(3)* in [FattorialeRec.java](#). ■

### 5.1.2 Quadrato

Riprendiamo un esempio già trattato quando abbiamo discusso gli invarianti:

**Esempio 8** (La funzione quadrato) Si può definire ricorsivamente il quadrato di un numero naturale mediante le clausole:

$$\begin{aligned} q(0) &= 0 \\ q(n) &= q(n - 1) + 2 * (n - 1) + 1 \end{aligned}$$

Vediamo che le clausole precedenti definiscono effettivamente la funzione desiderata, dimostrando per induzione che la proprietà  $q(n) = n * n$  è vera per ogni valore di  $n$ :  
(Base dell'induzione)

$$\begin{aligned} q(0) &= 0 && \text{(per definizione)} \\ &= 0^2 \end{aligned}$$

(Passo induttivo)

$$\begin{aligned} q(n) &= q(n - 1) + 2 * (n - 1) + 1 && \text{(per definizione)} \\ &= (n - 1)^2 + 2 * (n - 1) + 1 && \text{(per ipotesi induttiva)} \\ &= n^2 && \text{(per proprietà algebriche)} . \end{aligned}$$

Si osservi che le clausole della definizione ricorsiva della funzione  $q(n)$  consentono anche di calcolare il valore di questa funzione per un valore arbitrario dell'argomento. Per esempio:

$$\begin{aligned} q(5) &= q(4 + 1) \\ &= q(4) + 2 * 4 + 1 \\ &= q(3 + 1) + 2 * 4 + 1 \\ &= q(3) + 2 * 3 + 1 + 2 * 4 + 1 \\ &= q(2 + 1) + 2 * 3 + 1 + 2 * 4 + 1 \\ &= q(2) + 2 * 2 + 1 + 2 * 3 + 1 + 2 * 4 + 1 \\ &= q(1 + 1) + 2 * 2 + 1 + 2 * 3 + 1 + 2 * 4 + 1 \\ &= q(1) + 2 * 1 + 1 + 2 * 2 + 1 + 2 * 3 + 1 + 2 * 4 + 1 \\ &= q(0 + 1) + 2 * 1 + 1 + 2 * 2 + 1 + 2 * 3 + 1 + 2 * 4 + 1 \\ &= q(0) + 2 * 0 + 1 + 2 * 1 + 1 + 2 * 2 + 1 + 2 * 3 + 1 + 2 * 4 + 1 \\ &= 0 + 2 * 0 + 1 + 2 * 1 + 1 + 2 * 2 + 1 + 2 * 3 + 1 + 2 * 4 + 1 \\ &= 1 + 2 + 1 + 4 + 1 + 6 + 1 + 8 + 1 \\ &= 25 . \end{aligned}$$

**Esempio 9** Dato un insieme  $A$  ed una funzione  $f : A \rightarrow A$ , si può definire una collezione numerabile di funzioni  $f^n$ , una per ogni  $n \in \mathbb{N}$ , per ricorsione come segue, per ogni opportuno argomento  $a$ :

$$\begin{aligned} f^0(a) &= a \\ f^n(a) &= f(f^{n-1}(a)) . \end{aligned}$$

Si può dimostrare che:

$$f^n(a) = \underbrace{f(\dots f(a) \dots)}_{n \text{ volte}} .$$

**Esercizio 28** Dimostrare per induzione che la funzione definita dalle clausole:

$$\begin{aligned} f(0, y, z) &= z \times y \\ f(x, y, z) &= z + f(x - 1, y, z) \end{aligned} \quad (x > 0)$$

è tale che, per ogni  $x, y, z \in \mathbb{N}$ :

$$f(x, y, z) = z \times (x + y)$$

■

**Esercizio 29** Dimostrare per induzione che la funzione definita dalle clausole:

$$\begin{aligned} f(0) &= 1 \\ f(n) &= f(n - 1) + f(n - 1) \end{aligned} \quad (n > 0)$$

è tale che, per ogni  $n \in \mathbb{N}$ :

$$f(n) = 2^n.$$

■

**Esercizio 30** Dimostrare per induzione che la funzione definita dalle clausole:

$$\begin{aligned} s(0) &= 1 \\ s(n) &= 2/s(n - 1) \end{aligned} \quad (n > 0)$$

è tale che, per ogni  $n \in \mathbb{N}$ ,  $s(n) \in \{1, 2\}$ .

Inoltre, verificare quale dei seguenti predicati possa essere vero:

- $P(m) \equiv ((m \text{ pari}) \Rightarrow s(m) = 1) \vee ((m \text{ dispari}) \Rightarrow s(m) = 2)$ ;
- $P(m) \equiv ((m \text{ pari}) \Rightarrow s(m) = 1) \wedge ((m \text{ dispari}) \Rightarrow s(m) = 2)$ .

■

**Esercizio 31** Dimostrare per induzione su  $n$  che  $2^{(n+1)} - 1 = \sum_{i=0}^n 2^i$ .

■

**Esercizio 32** Il numero  $A(n)$  di modi in cui  $n$  persone possono essere assegnate a  $n$  poltrone può essere definito ricorsivamente mediante le clausole:

$$\begin{aligned} A(1) &= 1 \\ A(n) &= A(n - 1) \times n \end{aligned}$$

Dimostrare per induzione che, per ogni  $n \geq 1$ ,  $A(n) = n!$ .

■

## 5.2 Ricorsione e correttezza parziale per induzione

Il programma `sXYRec.java` riformula ricorsivamente il problema SIDP della somma tra due naturali.

Segue la dimostrazione di correttezza parziale che consiste nel dimostrare  $\forall n, m \in \mathbb{N}. P(m, n)$ , in cui:

$$P(m, n) \equiv \text{sXY}(m, n) = m + n . \quad (5.8)$$

Procedendo per induzione sul valore del secondo argomento di `sXY`, possiamo dimostrare:

$$\forall m \in \mathbb{N}. (P(m, 0) \wedge (\forall n \in \mathbb{N}. (P(m, n) \Rightarrow P(m, n + 1)))) \Rightarrow (\forall m, n \in \mathbb{N}. P(m, n)) . \quad (5.9)$$

**Caso base.** Fissiamo  $m \in \mathbb{N}$ . Dimostrare  $P(m, 0)$  equivale a dimostrare `sXY(m, 0)`. Per definizione, quando il secondo argomento vale 0, il programma restituisce  $m$ . Quindi, `sXY(m, 0) = m = m + 0` è quel che attendiamo.

**Caso induttivo.** Fissiamo  $m \in \mathbb{N}$  ed  $n \in \mathbb{N} \setminus \{0\}$ . Lo scopo è dimostrare  $P(m, n+1)$ , che equivale a  $sXY(m, n+1) = m + n + 1$ , assumendo  $P(m, n)$ , che equivale a  $sXY(m, n) = m + n$ .

Guardiamo al codice di [sXYRec.java](#). Richiamare  $sXY(m, n+1)$  equivale a voler restituire il risultato dell'espressione  $1 + sXY(x, y - 1)$ ; nella quale il parametro formale  $x$  assume il valore  $m$  ed il parametro formale  $y$  assume il valore  $n$ . Quindi, lo scopo è valutare l'espressione  $1 + sXY(m, (n+1)-1)$  che è uguale a  $1 + sXY(m, n)$ . Per ipotesi induttiva, sappiamo che  $sXY(m, n) = m + n$ , valore che possiamo sostituire a  $sXY(m, (n+1)-1)$ , ottenendo  $1 + m + n$ . Quindi,  $sXY(m, n+1) = m + n + 1$ . Il processo formale, senza prosa:

$$\begin{aligned}sXY(m, n+1) &= 1 + sXY(m, (n+1)-1) \\&= 1 + sXY(m, n) \\&= 1 + (m + n) \\&= m + (n + 1) .\end{aligned}$$

**Esercizio 33 (Dimostrazioni di base della correttezza parziale per induzione)** • Scrivere un algoritmo ricorsivo che risolva il problema MSDP, ovvero la moltiplicazione tra due numeri naturali, usando solo somme e predecessore. Dimostrare la sua correttezza parziale per induzione.

([mXYRec.java](#) è una possibile soluzione.)

- Scrivere un algoritmo ricorsivo che risolva il problema PPID, ovvero che, dati due numeri naturali  $a$  e  $b$ , calcoli il valore di  $a$  elevato a  $b$ , sotto l'ipotesi di saper solo calcolare il prodotto tra due numeri ed il predecessore di un numero. Dimostrare la sua correttezza parziale per induzione.

([eXYRec.java](#) è una soluzione possibile.)

■

### 5.2.1 Sommatoria di un segmento di naturali

Dimostriamo la correttezza del seguente metodo sorgente:

```
2 public static int sommatoriaTraZeroEd(int x) {
3 if (x == 0)
4 return 0;
5 else
6 return x + sommatoriaTraZeroEd(x - 1);
```

per induzione sul valore assunto dal parametro formale. Ovvero, se fissiamo  $\forall n \in \mathbb{N}. P(n)$ , in cui:

$$P(n) \equiv \text{sommatoriaTraZeroEd}(n) = \sum_{i=0}^n i , \quad (5.10)$$

procediamo per induzione sul valore dell'argomento di `sommatoriaTraZeroEd`, per dimostrare:

$$(P(0) \wedge (\forall n \in \mathbb{N}. (P(n) \Rightarrow P(n+1)))) \Rightarrow (\forall n \in \mathbb{N}. P(n)) . \quad (5.11)$$

**Caso base.** Dimostrare  $P(0)$  equivale a dimostrare  $\text{sommatoriaTraZeroEd}(0) = 0$ . Quando il parametro formale  $x$  vale 0, da un lato il metodo restituisce 0, dall'altro  $\sum_{i=0}^0 i = 0$ , quindi  $\text{sommatoriaTraZeroEd}(0) = 0 = \sum_{i=0}^0 i$ .

**Caso induttivo.** Sia  $n \in \mathbb{N} \setminus \{0\}$ . Lo scopo è dimostrare  $P(n+1)$ , che equivale a  $\text{sommatoriaTraZeroEd}(n+1) = \sum_{i=0}^{n+1} i$ , assumendo  $P(n)$ , che equivale a  $\text{sommatoriaTraZeroEd}(n) = \sum_{i=0}^n i$ .

Guardiamo al codice sorgente. Richiamare  $\text{sommatoriaTraZeroEd}(n+1)$  equivale a voler restituire il risultato dell'espressione  $x + \text{sommatoriaTraZeroEd}(x - 1)$  nella quale il parametro formale  $x$  assume il valore  $n + 1$ . Quindi, lo scopo è valutare l'espressione  $(n+1) + \text{sommatoriaTraZeroEd}((n+1)-1)$  che è uguale a  $(n+1) + \text{sommatoriaTraZeroEd}(n)$ . Per ipotesi induttiva,  $\text{sommatoriaTraZeroEd}(n) = \sum_{i=0}^n i$ , espressione che possiamo sostituire a  $\text{sommatoriaTraZeroEd}((n+1)-1)$ , ottenendo  $(n+1) + \sum_{i=0}^n i = \sum_{i=0}^{n+1} i$ . Quindi,

sommatoriaTraZeroEd( $n + 1$ ) =  $\sum_{i=0}^{n+1} i$ . Il processo formale, senza prosa, è il seguente:

$$\begin{aligned}\text{sommatoriaTraZeroEd}(n + 1) &= (n + 1) + \text{sommatoriaTraZeroEd}(n) \\ &= (n + 1) + \sum_{i=0}^n \\ &= (n + 1) + n + (n - 1) + \dots + 2 + 1 + 0 \\ &= \sum_{i=0}^{n+1} i .\end{aligned}$$

La classe `SommatoriaPrimiInteriRec.java` include sia il metodo appena analizzato, sia una versione alternativa che sfrutta l'espressione condizionale disponibile in Java.

### 5.2.2 Lettura e stampa di una sequenza di valori

Lo scopo di questa sezione è insistere sul fatto che le dimostrazioni di correttezza parziale si possono applicare anche a metodi ricorsivi che eseguono operazioni di input ed output di dati.

La classe `LeggeStampaEnneInteriRec.java` include il metodo ricorsivo:

```

2 public static void leggeStampaInteri(int x) {
3 if (x == 0) {
4 } else {
5 leggeStampaInteri(x - 1);
6 int numeroLetto = SIn.readInt();
7 System.out.println(numeroLetto);
8 }
}
```

che, fissato un valore  $n \geq 0$  per  $x$ , legge da tastiera e stampa su schermo una sequenza formata da  $n$  valori interi.

Per induzione sul valore  $n$ , assunto da  $x$ , dimostriamo  $\forall n \in \mathbb{N}. P(n)$ , in cui:

$$P(i) \equiv (\forall k. 0 \leq k < i \Rightarrow \text{il } k\text{-esimo intero è stato letto e stampato}) , \quad (5.12)$$

ovvero, dimostriamo:

$$(P(0) \wedge (\forall i \in \mathbb{N}. (P(i) \Rightarrow P(i + 1)))) \Rightarrow (\forall n \in \mathbb{N}. P(n)) . \quad (5.13)$$

**Caso base.** Sia  $i = 0$ . Per definizione del codice, nel caso in questione, `leggeStampaInteri(0)` non stampa nulla. Per definizione, l'istanza  $P(0)$  è  $(\forall k. 0 \leq k < 0 \Rightarrow \text{il } k\text{-esimo intero è stato letto e stampato})$ . Il predicato è vero perché la premessa  $0 \leq k < 0$  dell'implicazione è falsa per ogni valore di  $k$  siccome l'intervallo descritto è vuoto.

**Caso induttivo.** Sia  $i \in \mathbb{N}$ . Lo scopo è dimostrare:

$$P(i + 1) \equiv (\forall k. 0 \leq k < i + 1 \Rightarrow \text{il } k\text{-esimo intero è stato letto e stampato}) ,$$

assumendo che sia vero:

$$P(i) \equiv (\forall k. 0 \leq k < i \Rightarrow \text{il } k\text{-esimo intero è stato letto e stampato}) .$$

Guardiamo al codice sorgente. Richiamare `leggeStampaInteri( $i + 1$ )` equivale ad interpretare le linee 4, 5 e 6:

- al termine della linea 4 di `leggeStampaInteri( $i + 1$ )`, avendo richiamato `leggeStampaInteri( $i$ )`, per ipotesi induttiva, abbiamo letto e stampato una sequenza  $v_0, v_1, \dots, v_{i-1}$  di valori interi, perché, per ipotesi induttiva vale  $P(i)$  che può essere “espanso” come:

lo 0-esimo intero è stato letto e stampato  $\wedge$   
 il 1-mo intero è stato letto e stampato  $\wedge$   
 il 2-do intero è stato letto e stampato  $\wedge$   
 $\vdots$   
 l'( $i - 2$ )-esimo intero è stato letto e stampato  $\wedge$   
 l'( $i - 1$ )-esimo intero è stato letto e stampato .

- Al termine della linea 5 di `leggeStampaInteri(i + 1)` è stato letto un valore che segue tutti i precedenti letti e stampati. Ovvero è stato letto  $v_i$ .
- Al termine della linea 6 di `leggeStampaInteri(i + 1)` è stato stampato un valore che segue tutti i precedenti letti e stampati. Ovvero è stato stampato  $v_i$ . Quindi, globalmente, siamo giunti nella situazione

lo 0-esimo intero è stato letto e stampato  $\wedge$   
 il 1-mo intero è stato letto e stampato  $\wedge$   
 il 2-do intero è stato letto e stampato  $\wedge$   
 :  
 l' $(i - 2)$ -esimo intero è stato letto e stampato  $\wedge$   
 l' $(i - 1)$ -esimo intero è stato letto e stampato  $\wedge$   
 l' $i$ -esimo intero è stato letto e stampato ,

che si può riassumere proprio con  $P(i + 1) \equiv (\forall i. 0 \leq i < i + 1 \Rightarrow \text{l}'i\text{-esimo intero è stato letto e stampato})$ .

**Esercizio 34 (Correttezza parziale per induzione)** Fissato un valore  $n \geq 1$ , usare il principio di induzione per dimostrare la correttezza parziale dei metodi ricorsivi nella seguente classe `EnneInteriMaxRec.java` il cui metodo legge  $n$  interi, e ne restituisce il massimo.

### 5.2.3 Ricorsione “controvariante” e metodi involucro

È certamente possibile immaginare di poter scrivere una funzione ricorsiva come segue:

$$\begin{aligned} \text{meno}(m, n, n) &= m \\ \text{meno}(m, n, p) &= \text{meno}(m, n, p + 1) - 1 & p < n . \end{aligned}$$

Il nome suggerisce che essa calcoli la differenza tra  $m$  ed  $n$ , supponendo  $m, n \in \mathbb{N}$  e  $m \geq n$ . Il predicato con cui dimostrarlo è:

$$P(i) \equiv \text{meno}(m, n, n - i) = m - i .$$

La dimostrazione del caso base diventa:

$$\begin{aligned} P(0) &\Leftrightarrow \text{meno}(m, n, n - 0) = m - 0 \\ &\Leftrightarrow \text{meno}(m, n, n) = m \end{aligned}$$

vero per definizione.

Per il caso induttivo assumiamo che valga  $P(i - 1)$  e proseguiamo con le seguenti equivalenze logiche:

$$\begin{aligned} P(i - 1) &\Leftrightarrow \text{meno}(m, n, n - (i - 1)) = m - (i - 1) \\ &\Leftrightarrow \text{meno}(m, n, n - (i - 1)) - 1 = m - (i - 1) - 1 \\ &\Leftrightarrow \text{meno}(m, n, n - i) = m - (i - 1) - 1 \\ &\Leftrightarrow \text{meno}(m, n, n - i) = m - i + 1 - 1 \\ &\Leftrightarrow \text{meno}(m, n, n - i) = m - i \\ &\Leftrightarrow P(i) . \end{aligned}$$

Se ne deduce che per calcolare la differenza tra  $m$  ed  $n$  occorre usare  $\text{meno}(m, n, 0)$ . Dal punto di vista della programmazione in Java nel caso illustrato è fortemente suggerito l'uso di due metodi nella stessa classe come segue:

```

public static int meno(int m, int n) {
 2 if (m < n)
 3 return 0;
 4 else
 5 return meno(m, n, 0);
 6 }

 8 private static int meno(int m, int n, int p) {
 9 if (p == n)
 10 return m
 11 else
 12 return meno(m, n, p + 1) - 1;
}

```

su cui valgono alcune osservazioni:

- Entrambi i metodi hanno lo stesso nome, ma differiscono per numero di parametri formali. Ovvero la loro *signature* che, per definizione, è composta da nome del metodo e dall'elenco dei tipi dei parametri formali, è differente.

Java permette la coesistenza di metodi con *signature* diverse nella stessa classe.

Se due metodi della stessa classe hanno *signature* diversa, ma nome identico, allora si dice che sono *overloaded*.

- Il metodo con *signature* `meno(int, int)` è pubblico, ovvero utilizzabile da una qualsiasi classe. Quello con *signature* `meno(int, int, int)` è privato, ovvero utilizzabile (richiamabile) solo da metodi nella classe in cui `meno(int, int, int)` stesso sia contenuto.

La scelta sulla restrizione di visibilità di `meno(int, int, int)` è imposta dal fatto che esso restituisca un valore corretto solo se richiamato come `meno(m, n, 0)`, per qualsiasi coppia di naturali  $m$  ed  $n$ , come avviene in `meno(int, int)`. Equivalentemente, si impedisce che `meno(int, int, int)` sia richiamabile da un qualsiasi metodo diverso da `meno(int, int)`, usando un valore diverso da zero come terzo parametro attuale.

- Il *wrapper* è stato utilizzato per discriminare i casi in cui non sia necessario innescare il meccanismo ricorsivo di calcolo della differenza.

**Esercizio 35 (Funzioni ricorsive controvarianti.)** 1. Data la funzione:

$$\begin{aligned} \text{piu}(m, n, n) &= m \\ \text{piu}(m, n, p) &= \text{piu}(m, n, p + 1) + 1 & p < n \end{aligned}$$

dimostrare che essa calcola la funzione descritta dal predicato:

$$P(i) \equiv \text{piu}(m, n, n - i) = m + i ,$$

scrivere il metodo e *wrapper* corrispondenti.

2. Data la funzione:

$$\begin{aligned} \text{per}(m, n, n) &= 0 \\ \text{per}(m, n, p) &= \text{per}(m, n, p + 1), m, 0 & p < n \end{aligned}$$

dimostrare che essa calcola la funzione descritta dal predicato:

$$P(i) \equiv \text{per}(m, n, n - i) = m * i .$$

Scrivere anche metodo e *wrapper* corrispondenti.

3. Data la funzione:

$$\begin{aligned} \text{exp}(m, n, n) &= 1 \\ \text{exp}(m, n, p) &= \text{per}(\text{exp}(m, n, p + 1), m, 0) & p < n \end{aligned}$$

dimostrare che essa calcola la funzione descritta dal predicato:

$$P(i) \equiv \text{exp}(m, n, n - i) = m^i .$$

Scrivere anche metodo e *wrapper* corrispondenti.

4. Data la funzione:

$$\begin{aligned} \text{quoziante}(D, d, r) &= 0 & r > D \\ \text{quoziante}(D, d, r) &= \text{quoziante}(D, d, r + d) + 1 & r \leq D \end{aligned}$$

convincersi che calcoli il quoziente della divisione intera tra  $D$  e  $d$ . Scrivere il metodo ed il *wrapper* corrispondenti. ■

### 5.3 Ricorsione di coda

Questo argomento è inserito per completezza. La ragione per parlare di ricorsione di coda è una possibile osservazione sul costo che gli algoritmi ricorsivi hanno, in termini di occupazione di memoria. In generale, al crescere della dimensione, opportunamente misurata, dei dati di input, cresce, anche più che proporzionalmente, lo spazio necessario ad allocare la pila di *frame*.

È possibile non rinunciare né alla definizione ricorsiva di metodi, né ad un buon utilizzo della memoria mirato a contenere l'estendersi del *frame stack*, conseguente alle chiamate ricorsive.

La strategia consiste nello scrivere metodi *ricorsivi di coda*:

Un metodo ricorsivo è detto “di coda” se è definito in modo che ogni richiamo ricorsivo non sia seguito da alcuna ulteriore istruzione da interpretare.

Il confronto tra metodi ricorsivi che calcolino  $n!$  dovrebbe chiarire il significato della definizione.

Il primo metodo è quello già visto:

```

1 public static int fatt (int x) {
2 if (x == 0)
3 return 1;
4 else
5 return x * fatt(x - 1);
}
```

Uno di coda equivalente è:

```

1 public static int fattCoda (int x, int f) {
2 if (x == 0)
3 return f;
4 else
5 return fatt(x - 1, x * f);
6 }
```

Il metodo `fattCoda` ha un parametro in più il cui scopo è accumulare il risultato, raccogliendolo dal chiamante, per passarlo al chiamato, dopo le opportune manipolazioni. Il nuovo parametro sposta il calcolo della moltiplicazione per  $x$  da dopo il richiamo ricorsivo, come in `fatt` al momento in cui si valutano i parametri attuali.

Una dimostrazione di correttezza parziale è un modo per “vedere” il meccanismo all’opera. Dimostriamo, quindi,  $\forall m \in \mathbb{N}, n \in \mathbb{N} \setminus \{0\}. P(m, n)$  in cui:

$$P(m, n) \equiv \text{fattCoda}(m, n) = m! * n . \quad (5.14)$$

Una volta fissato  $n \geq 1$ , procediamo per induzione sul valore  $m$ , assunto dal primo parametro formale di `fattCoda`, per dimostrare:

$$\forall n \in \mathbb{N} \setminus \{0\}. (P(0, n) \wedge (\forall m \in \mathbb{N}. P(m, n) \Rightarrow P(m + 1, n))) \Rightarrow (\forall m \in \mathbb{N}, n \in \mathbb{N} \setminus \{0\}. P(m, n)) . \quad (5.15)$$

**Caso base.** Per definizione, `fattCoda(0, n)` restituisce  $n = 1 * n = 0! * 1$ .

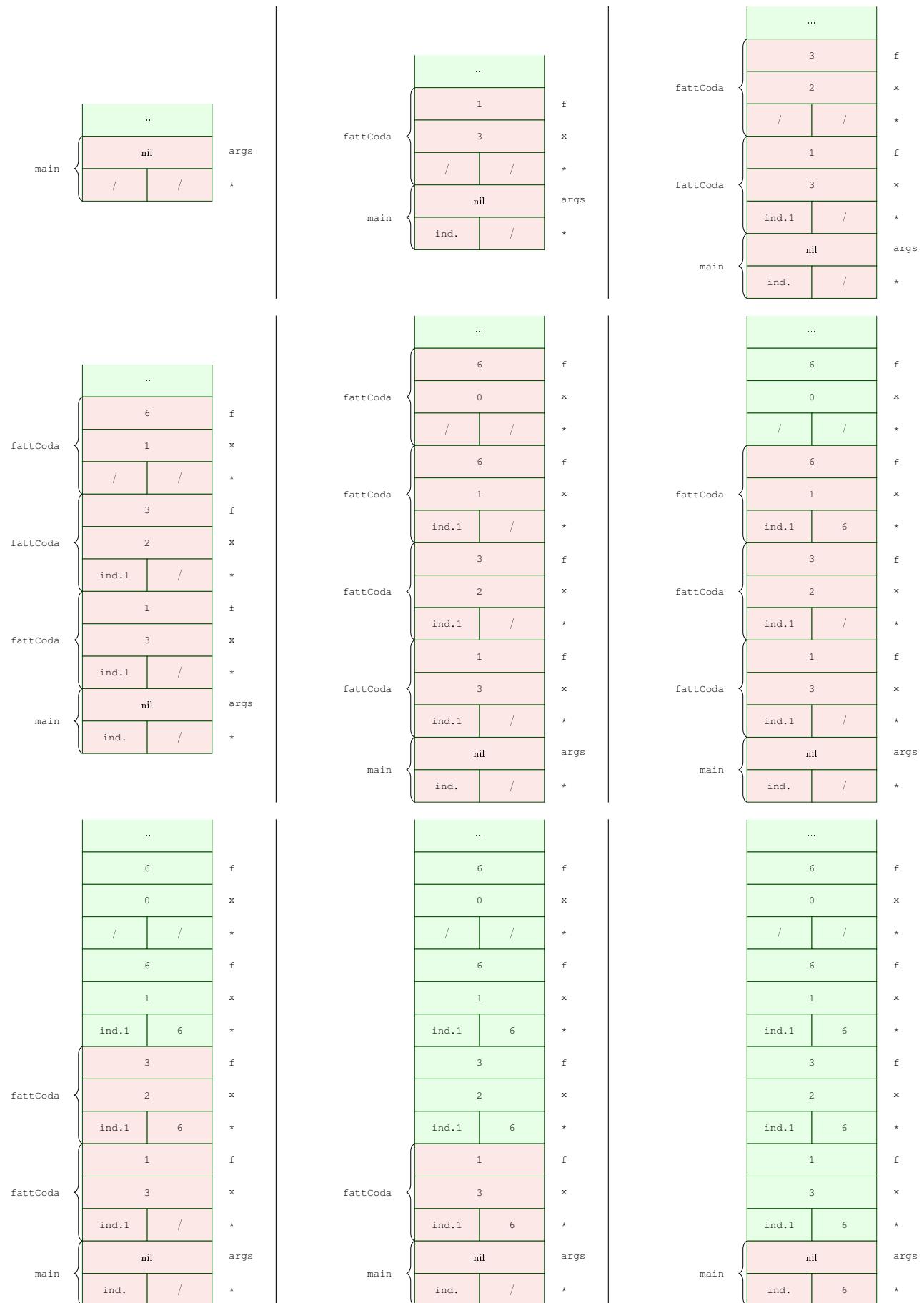
**Caso induttivo.** Per definizione, `fattCoda(m + 1, n)` è equivalente a calcolare `fattCoda((m + 1) - 1, (m + 1) * n)`. Per ipotesi induttiva `fattCoda((m + 1) - 1, (m + 1) * n) = (m + 1)! * n`, quindi:

$$\begin{aligned} \text{fattCoda}(m + 1, n) &= \text{fattCoda}((m + 1) - 1, (m + 1) * n) \\ &= (m + 1)! * n . \end{aligned}$$

In particolare, otteniamo `fattCoda(m, 1) = m!`.

`FattorialeRecCoda.java` è una classe completa per cominciare a sperimentare l’uso di metodi ricorsivi di coda.

Nonostante l’introduzione del nuovo parametro formale `f`, l’espansione del *frame stack* di `fattCoda` è analoga a, se non peggiore di, quello per `fatt`. Ad esempio, vediamo l’evoluzione del *frame stack* relativo ad una chiamata `fattCoda(3, 1)` in ipotetico, metodo `main`:



Riguardo ai vantaggi offerti dalla ricorsione di coda, quel che occorre realizzare è quanto segue:

- il solo impostare `fattCoda` come metodo di coda non è sufficiente ad ottenere una gestione parsimoniosa della memoria.

- Quel che conta è che metodi ricorsivi di coda come, ad esempio, `fattCoda` sono *automaticamente* trasformabili in programmi iterativi equivalenti che, proprio perché iterativi, non estendono l'occupazione della memoria da parte del *frame stack* oltre il necessario.

Il metodo iterativo, che *automaticamente* possiamo ricavare da `fattCoda` è:

```

2 public static int fattIterDaCoda (int x, int f) {
3 while (x > 0) {
4 f = x * f;
5 x = x - 1;
6 }

```

Osserviamo che:

- abbiamo trasformato il passaggio (del valore) di  $x - 1$  al parametro formale  $x$  in un'assegnazione  $x = x - 1$ ,
- abbiamo trasformato il passaggio (del valore) di  $x * f$  al parametro formale  $f$  in un'assegnazione  $f = x * f$ ,
- siccome il valore di  $x$  da cui deve dipendere l'espressione  $x * f$  deve essere quello prima del decremento  $x - 1$ , l'ordine delle assegnazioni è invertito, rispetto a quello dei parametri.

Dovrebbe essere intuitivo sia come procedere in maniera automatica, sia che vale la pena definire metodi ricorsivi di coda in modo che il parametro che guida la ricorsione sia l'ultimo elencato. Con questo ultimo accorgimento, non c'è il problema di dover ordinare le assegnazioni diversamente dagli argomenti.

### Esercizio 36 (Ricorsione di coda)

- Scrivere una classe con un metodo ricorsivo di coda che, presi due numeri naturali  $m$  ed  $n$ , calcoli la loro somma per incrementi e decrementi successivi di una unità.

(Soluzione possibile [sXYRecCoda.java](#).)

- Scrivere una classe con un metodo ricorsivo che calcoli il prodotto tra due numeri naturali.

(Soluzione possibile [mXYRecCoda.java](#).)

- Scrivere una classe con un metodo ricorsivo di coda che calcoli l'elevamento a potenza di un numero naturale per un altro naturale.

(Soluzione possibile [eXYRecCoda.java](#).)

- Scrivere una classe con un metodo ricorsivo di coda che, presi due numeri naturali  $m$  ed  $n$ , calcoli il resto della divisione intera di  $m$  per  $n$  per sottrazioni successive.

(Soluzione possibile [RestoRecCoda.java](#).)

- Scrivere una classe con un metodo ricorsivo che, fissato un valore  $n \geq 1$ , legga  $n$  interi e ne calcoli la media. Affinché il programma sia ricorsivo di coda va organizzato in modo che sia l'ultimo passo della ricorsione a restituire la media.

(Soluzione possibile [MediaRecCoda.java](#).)

■

## 5.4 Induzione forte e funzioni ricorsive parallelizzabili

Il principio di induzione forte, o completa, è uno schema di dimostrazione equivalente a quello che conosciamo, ma più flessibile. Formalmente, il principio si enuncia come segue:

$$(P(0) \wedge (\forall i, k \in \mathbb{N}. (k < i \Rightarrow P(k)) \Rightarrow P(i))) \Rightarrow (\forall n \in \mathbb{N}. P(n)) . \quad (5.16)$$

Esso permette ragionamenti induttivi più “flessibili” perché applicabili a funzioni ricorsive nelle quali l'argomento del richiamo ricorsivo può calare arbitrariamente, ovviamente senza diventare negativo.

### 5.4.1 Quoziente tra naturali per sottrazioni iterate

Un problema noto che offre l'opportunità di usare il principio di induzione forte è il calcolo del quoziente tra naturali, ottenuto per sottrazioni successive del divisore dal dividendo ed impostando ricorsivamente la definizione della funzione calcolata.

Siano quindi dati il dividendo  $D$  ed il divisore  $d$ . Allora, se  $D < d$  non è possibile fare alcuna divisione intera di  $d$  elementi presi da  $D$  ed il quoziente è per forza nullo.

Supponiamo, invece, d'avere almeno  $d$  elementi da distribuire in  $D$ . Allora, possiamo conteggiare una distribuzione in più rispetto a quelle che rimangono da fare con  $D - d$  elementi.

Formalmente, caso base ed induttivo, qui sopra esposti intuitivamente, diventano la seguente definizione ricorsiva:

$$\begin{aligned} \text{quoziente}(D, d) &= 0 && \text{se } D < d \\ \text{quoziente}(D, d) &= 1 + \text{quoziente}(D - d, d) && \text{se } D \geq d . \end{aligned}$$

Il predicato proposto per descrivere la correttezza parziale è:

$$P(x, d) \equiv \exists r. x = d * \text{quoziente}(x, d) + r .$$

L'istanza  $P(0, d)$  è vera perché:

$$\begin{aligned} P(0, d) &\Leftrightarrow \exists r. 0 = d * \text{quoziente}(0, d) + r && \text{siccome } 0 \leq d \\ &\Leftrightarrow \exists r. 0 = d * 0 + r \\ &\Leftrightarrow 0 = d * 0 + 0 . \end{aligned}$$

Supponiamo  $D \geq d$ . La tesi è vera grazie alle seguenti equivalenze logiche:

$$\begin{aligned} P(D, d) &\Leftrightarrow \exists r. D = d * \text{quoziente}(D, d) + r \\ &\Leftrightarrow \exists r. D = d * (1 + \text{quoziente}(D - d, d)) + r && \text{per definizione} \\ &\Leftrightarrow \exists r. D = d + d * \text{quoziente}(D - d, d) + r \\ &\Leftrightarrow \exists r. D = d * \text{quoziente}(D - d, d) + r + d \\ &\Leftrightarrow \exists r. D - d = d * \text{quoziente}(D - d, d) + r \\ &\Leftrightarrow P(D - d, d) \end{aligned}$$

in cui l'ultimo predicato è vero perché vale l'ipotesi induttiva forte:

$$\forall k \in \mathbb{N}. k < D \Rightarrow P(k, d)$$

di cui sfruttiamo l'istanza  $D - d < D$ , rendendo vero  $P(D - d, d)$ .

#### 5.4.2 Resto del quoziente tra naturali per sottrazioni iterate

Un ulteriore problema noto che offre l'opportunità di usare il principio di induzione forte è il calcolo del resto della divisione intera tra naturali, ottenuta per sottrazioni successive del divisore dal dividendo ed impostando ricorsivamente la definizione della funzione calcolata.

Siano quindi dati il dividendo  $D$  ed il divisore  $d$ . Allora, se  $D < d$  non è possibile fare alcuna divisione intera di  $d$  elementi presi da  $D$  ed il resto è per forza  $D$ .

Supponiamo, invece, d'avere almeno  $d$  elementi da distribuire in  $D$ . Allora, il resto sarà ottenuto continuando a distribuire, cioè a dividere quanto rimasto in  $D$  fra  $d$  parti.

Formalmente, caso base ed induttivo, qui sopra esposti intuitivamente, diventano la seguente definizione ricorsiva:

$$\begin{aligned} \text{resto}(D, d) &= D && \text{se } D < d \\ \text{resto}(D, d) &= \text{resto}(D - d, d) && \text{se } D \geq d . \end{aligned}$$

Il predicato proposto per descrivere la correttezza parziale è:

$$P(x, d) \equiv \exists q. x = d * q + \text{resto}(x, d) .$$

L'istanza  $P(0, d)$  è vera perché:

$$\begin{aligned} P(0, d) &\Leftrightarrow \exists q. 0 = d * q + \text{resto}(0, d) && \text{siccome } 0 \leq d \\ &\Leftrightarrow \exists q. 0 = d * q + 0 \\ &\Leftrightarrow 0 = d * 0 + 0 . \end{aligned}$$

Supponiamo  $D \geq d$ . Siccome  $D - d < D$ , usando l'ipotesi di'induzione forte:

$$\forall k \in \mathbb{N}. k < D \Rightarrow P(k, d)$$

abbiamo in particolare che  $P(D - d, d)$  è vero. Quindi valgono le seguenti equivalenze logiche:

$$\begin{aligned}
 P(D - d, d) &\Leftrightarrow \exists q. D - d = d * q + \text{resto}(D - d, d) \\
 &\Leftrightarrow \exists q. D = d * q + d + \text{resto}(D - d, d) \\
 &\Leftrightarrow \exists q. D = d * (q + 1) + \text{resto}(D - d, d) \\
 &\Leftrightarrow \exists q'. D = d * q' + \text{resto}(D - d, d) && \text{per definizione} \\
 &\Leftrightarrow \exists q. D = d * q + \text{resto}(D, d) \\
 &\Leftrightarrow P(D, d)
 \end{aligned}$$

in cui l'ultimo predicato vero è quel che vogliamo dimostrare.

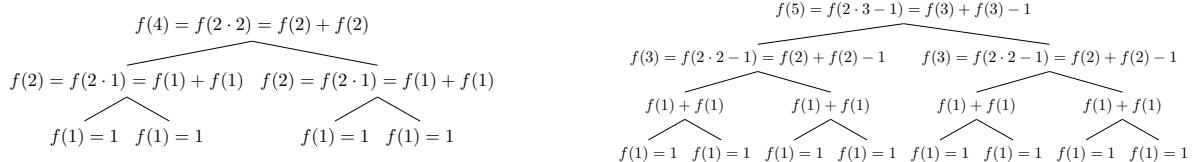
### 5.4.3 Funzione identità

La seguente funzione

$$\begin{aligned}
 f(0) &= 0 \\
 f(1) &= 1 \\
 f(2m) &= f(m) + f(m) && (m > 0) \\
 f(2m - 1) &= f(m) + f(m) - 1 && (m > 1)
 \end{aligned}$$

tra numeri naturali costituisce uno schema di riferimento per impostare programmi ricorsivi che, in linea di principio, possano essere interpretati parallelamente.

Tra poco dimostreremo che  $f$  esegue una banale operazione di conteggio, siccome, per ogni  $x$ ,  $f(x) = x$ . L'aspetto rilevante, però, è lo schema in accordo col quale essa opera e che può essere rappresentato con strutture ad albero come le seguenti:



nelle quai ogni sotto-albero produce il risultato indipendentemente dagli altri sotto-alberi.

Dimostriamo ora la proprietà  $\forall n. P(n)$ , in cui:

$$P(n) \equiv f(n) = n . \quad (5.17)$$

Procedendo per induzione forte sul valore dell'argomento di  $f$ , possiamo dimostrare:

$$(P(0) \wedge (\forall i, k \in \mathbb{N}. (k < i \Rightarrow P(k)) \Rightarrow P(i))) \Rightarrow (\forall n \in \mathbb{N}. P(n)) . \quad (5.18)$$

**Caso base.** Per definizione, dimostrare  $P(0)$  equivale a dimostrare  $f(0) = 0$ , equivalenza vera per definizione di  $f$ .

**Caso induttivo.** Siccome ragioniamo per induzione completa, assumiamo che valga  $k < i \Rightarrow P(k)$  per un valore fissato di  $i \geq 1$ . Necessariamente,  $i$  è pari o dispari. Se  $i \geq 1$  è pari, allora esiste  $m > 0$  tale che  $i = 2m$ . Se  $i \geq 1$  è dispari, allora esiste  $m \geq 1$  tale che  $i = 2m - 1$ . In entrambi i casi,  $m < i$  e per ipotesi induttiva vale  $P(m)$ , ovvero  $f(m) = m$ . Ma questa equivalenza, permette di concludere che vale  $P(i)$ , sia con  $i$  pari, sia con  $i$  dispari:

- se  $i \geq 1$  è pari, da  $f(m) = m$  otteniamo  $f(i) = f(2m) = 2f(m) = 2m = i$ ,
- se  $i \geq 1$  è dispari, da  $f(m) = m$  otteniamo  $f(i) = f(2m - 1) = 2f(m) - 1 = 2m - 1 = i$ .

### 5.4.4 Funzione successore dicotomica

Sia  $g$  la seguente funzione:

$$\begin{aligned}
 g(0) &= 1 \\
 g(n) &= g(\lceil n/2 \rceil - 1) + g(\lfloor n/2 \rfloor) .
 \end{aligned}$$

Al contrario della funzione  $f$  della precedente Sottosezione 5.4.3,  $g$  non sfrutta il così detto meccanismo di *pattern matching* per stabilire quale tra le due clausole eseguire. Intendiamo che non si demanda al momento dell'applicazione

il problema di discriminare se l'argomento sia pari o dispari. Al contrario, affida alle operazioni  $\lceil x \rceil$  e  $\lfloor x \rfloor$  il compito di calcolare il giusto valore dell'argomento da utilizzare nel sotto-problema ricorsivo.

Dimostriamo  $\forall n. Q(n)$  in cui:

$$Q(n) \equiv n \in \mathbb{N} \Rightarrow g(n) = n + 1 . \quad (5.19)$$

Procedendo per induzione forte sul valore dell'argomento di  $g$ , possiamo dimostrare:

$$(Q(0) \wedge (\forall i, k \in \mathbb{N}. (k < i \Rightarrow Q(k)) \Rightarrow Q(i))) \Rightarrow (\forall n \in \mathbb{N}. Q(n)) . \quad (5.20)$$

**Caso base.** Per definizione, dimostrare  $Q(0)$  equivale a dimostrare  $g(0) = 1$ , equivalenza vera per definizione di  $g$ .

**Caso induttivo.** Siccome ragioniamo per induzione completa, assumiamo che valga  $k < i \Rightarrow Q(k)$  per un valore fissato di  $i > 0$ . Per definizione di  $\lceil \cdot \rceil$  e  $\lfloor \cdot \rfloor$  valgono le relazioni  $\lceil i/2 \rceil - 1 < i$  e  $\lfloor i/2 \rfloor < i$ . Per ipotesi induttiva, ne consegue che  $g(\lceil i/2 \rceil - 1) = \lceil i/2 \rceil - 1 + 1$  e  $g(\lfloor i/2 \rfloor) = \lfloor i/2 \rfloor + 1$ . Quindi:

$$\begin{aligned} g(i) &= g(\lceil i/2 \rceil - 1) + g(\lfloor i/2 \rfloor) && (\text{ipotesi induttiva}) \\ &= \lceil i/2 \rceil - 1 + 1 + \lfloor i/2 \rfloor + 1 \\ &= \lceil i/2 \rceil + \lfloor i/2 \rfloor + 1 && (\text{proprietà di } \lceil \cdot \rceil \text{ e } \lfloor \cdot \rfloor) \\ &= i + 1 . \end{aligned}$$

**Esercizio 37 (Funzioni che usano l'induzione forte per la correttezza parziale.)** 1. Data la funzione:

$$\begin{aligned} f(0) &= 0 \\ f(n) &= f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) \end{aligned}$$

in cui  $n \geq 0$ , dimostrare che essa calcola l'identità.

2. Data la funzione:

$$\begin{aligned} f(0) &= 1 \\ f(2m+1) &= 2 \cdot f(m) \\ f(2m) &= 2 \cdot f(m) - 1 \end{aligned}$$

in cui  $m \geq 0$ , dimostrare che essa calcola il successore.

3. Data la funzione:

$$\begin{aligned} f(l, l+1) &= 1 \\ f(l, r) &= f(l, \lfloor (l+r)/2 \rfloor) + f(\lceil (l+r)/2 \rceil, r) && (r > l+1) \end{aligned}$$

in cui  $l, r \geq 0$ , dimostrare che essa calcola  $r - l$ .

4. Data la funzione:

$$\begin{aligned} f(l, l) &= 1 \\ f(l, r) &= f(l, \lfloor (l+r)/2 \rfloor) + f(\lceil (l+r)/2 \rceil + 1, r) && (r > l) \end{aligned}$$

in cui  $l, r \geq 0$ , dimostrare che essa calcola  $r - l + 1$ .

5. Data la funzione:

$$\begin{aligned} f(l, l) &= 1 \\ f(l, r) &= 0 && (l > r) \\ f(l, r) &= f(l, \lfloor (l+r)/2 \rfloor - 1) + f(\lfloor (l+r)/2 \rfloor, \lceil (l+r)/2 \rceil) + f(\lceil (l+r)/2 \rceil + 1, r) && (l < r) \end{aligned}$$

in cui  $l, r \geq 0$ , dimostrare che essa calcola  $r - l + 1$  se  $r \geq l$ , mentre è 0 nel caso  $r < l$ .

**Esercizio 38 (Aritmetica dicotomica)** Ideare definizioni ricorsive in stile dicotomico della somma, differenza, moltiplicazione, quoziente e resto tra numeri interi.

([DicotomicaAritmetica.java](#) propone possibili soluzioni.)

□

**Esercizio 39 (Stampa dicotomica)** Ideare almeno un metodo ricorsivo per stampare i primi  $n$  numeri naturali.

([DicotomicaStampaSegmento.java](#) propone possibili soluzioni.) □

**Commento 1 (Alcune funzioni ricorsive classiche.)**

Due funzioni ricorsive classiche definite in base al principio di induzione forte sono:

- il massimo comun divisore. La classe [MCDRec.java](#) se confrontata con [MCD.java](#) evidenzia l'ovvietà della prova di correttezza della versione ricorsiva;
- la funzione che genera la sequenza di Fibonacci. Una classe con la sua possibile implementazione è [FibonacciRec.java](#) e fornisce una [Simulazione del frame stack di FibonacciRec.java](#). □

## 5.5 Funzioni ricorsive famose (NON nel programma didattico)

### 5.5.1 Coefficiente binomiale

[CoeffBinomialeRec.java](#) è una funzione interessante almeno perché conta il numero di combinazioni composte da  $k$  elementi presi tra  $n$ .

### 5.5.2 Torre di Hanoi

Il problema della [Torre di Hanoi su Wikipedia](#) ha una natura squisitamente combinatoria. Il metodo Java:

```

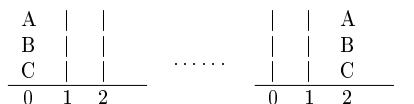
1 public static void muoviTorre (int n, int s, int d, int aus) {
2 if (n > 0) {
3 muoviTorre (n-1,s,aus,d);
4 muoviDisco(s,d);
5 muoviTorre (n-1,aus,d,s);
6 }
7 }
```

rappresenta una soluzione ricorsiva al problema, assumendo:

- l'esistenza di un metodo `muoviDisco` che realizzi effettivamente lo spostamento di un disco dal piolo sorgente  $s$  al piolo destinazione  $d$ ,
- che  $n$  indichi il numero di dischi impilati sul piolo sorgente  $s$ ;
- che  $d$  sia il piolo destinazione;
- che  $aus$  sia il piolo ausiliario, per gli spostamenti intermedi.

Il motivo per cui la descrizione dell'algoritmo è estremamente compatta dipende proprio dal fatto che il problema in cui occorra spostare  $n$  dischi è scomposto in due problemi più piccoli, ovvero che operano solo su  $n - 1$  dischi e che tali problemi più piccoli siano istanze della Torre di Hanoi stessa, ma con un uso differente dei pioli.

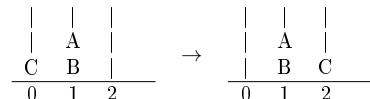
Leggiamo il meccanismo di soluzione assumendo  $n = 3$ . Lo scopo è passare dalla configurazione iniziale a sinistra a quella finale a destra, nella quale A è il disco con diametro inferiore, B quello intermedio e C il più grande:



Ragioniamo come se il disco C non esistesse. Sotto questa ipotesi, ci troviamo a dover risolvere la Torre di Hanoi con soli due dischi. In particolare, possiamo decidere di risolverlo, usando il piolo 2 come appoggio ed il 1 come piolo finale; risolviamo quindi il problema `muoviTorre(2, 0, 1, 2)`. Questo significa supporre di passare in un sol colpo dalla configurazione a sinistra, che è anche quella iniziale, a quella a destra, seguenti:



Nella configurazione illustrata ci rendiamo conto che il disco C è libero di essere mosso dalla sua sede al piolo 2, perché nulla lo impedisce. Possiamo, quindi passare dall'ultima configurazione che abbiamo supposto di sapere raggiungere a quella successiva, con C al piolo 2:



L'ultima configurazione ci ripropone di risolvere la Torre di Hanoi con due dischi che, dal piolo centrale 1 devono essere spostati al piolo 2, usando il piolo 0 come appoggio; risolviamo quindi il problema `muoviTorre(2, 1, 3, 1)`. Passiamo, quindi dall'ultima configurazione a quella finale:



Quando la soluzione alla Torre di Hanoi è ovvia? Quando c'è un solo disco da spostare dal piolo cui è infilato al piolo che *in quel frangente*, ovvero nella configurazione cui ci troviamo, assume il ruolo di destinazione.

**Esercizio 40** Dimostrare per induzione su  $m$  che `muoviTorre` esegue l'istruzione `muoviDisco(s, d)` per  $2^m - 1$  volte, supponendo che il valore iniziale di `n` sia  $m$ . ■

`TorreHanoi.java` è una classe che “completa” `muoviTorre`, stampando la sequenza di mosse per risolvere il problema della Torre di Hanoi.

**Esercizio 41** Considerare il seguente sorgente:

```

1 public static String muoviTorre(int n, int s, int aus, int d, String mosse) {
2
3 if (n == 1)
4 mosse = sposta(s, d, mosse);
5 else {
6 mosse = muoviTorre(n - 1, s, aus, d, mosse);
7 mosse = sposta(s, aus, mosse);
8 mosse = muoviTorre(n - 1, d, aus, s, mosse);
9 mosse = sposta(aus, d, mosse);
10 mosse = muoviTorre(n - 1, s, aus, d, mosse);
11 }
12 return mosse;
13 }
```

e, disegnando a mano il passaggio da una configurazione all'altra, convincersi che risolva il problema della Torre di Hanoi. La classe `TorreHanoiConfigurazioniNonOttimale.java` “completa” `muoviTorre` qui sopra definita così da stampare gradevolmente la sequenza di configurazioni.

È possibile immaginare di calcolare il numero di mosse usate da questo nuovo algoritmo? ■

### 5.5.3 Funzione di McCarty

`McCarthy91.java` interessante per il suo comportamento oscillatorio che costituisce una sfida nei confronti degli strumenti di verifica automatica della correttezza.

### 5.5.4 Funzione di Ackermann

`Ackermann.java` è interessante per i suoi legami con la teoria della computazione e per i valori che può assumere: essi crescono così rapidamente che per calcolarli si ottiene uno *stack overflow error*, ovvero si esaurisce lo spazio disponibile per il *frame stack*.



# Capitolo 6

## Programmazione con array

Il riferimento per la maggior parte degli argomenti trattati in questa parte di programma didattico è [SM14, Capitolo 6].

### 6.1 Una “scusa” per introdurre gli *array*

Abbiamo già incontrato il problema di riorganizzare i valori inizialmente contenuti in quattro variabili  $a$ ,  $b$ ,  $c$  e  $d$ , in modo che, al termine della riorganizzazione, il valore maggiore si trovi in  $d$ . Un modo compatto per descrivere la configurazione finale del problema è utilizzare nomi indicizzati per le variabili. Immaginando di ridenominare  $a$  come  $a_0$ ,  $b$  come  $a_1$ ,  $c$  come  $a_2$  e  $d$  come  $a_3$ , possiamo dire che, al termine, occorre che:

$$a_i \leq a_3 \quad \text{per ogni } 0 \leq i < 4 . \quad (6.1)$$

Una prima conseguenza è la compattezza della descrizione che sfrutta la variabilità dell’indice nell’intervallo dato.

Una seconda conseguenza è che l’intervallo entro cui  $i$  possa variare può essere arbitrario, anche se finito. Questo significa che potremmo immaginare di partire da un insieme di 10 variabili  $a_0, \dots, a_9$ , ciascuna con un valore, per arrivare allo stesso insieme di variabili in cui i valori iniziali siano stati risistemati in modo che:

$$a_i \leq a_9 \quad \text{per ogni } 0 \leq i < 10 . \quad (6.2)$$

La terza conseguenza è che la descrizione stessa della riorganizzazione dei valori diventa generalizzabile perché dipende dal valore massimo dei valori assumibili dall’indice che usiamo per identificare le variabili. Possiamo scrivere il seguente algoritmo:

```
// a0, ..., a9 contengono, ciascuna, un valore numerico
while (i < 9) do
 if ai > ai+1 then
 scambio dei valori in ai e ai+1
 incremento di i
 end if
end while
```

Java fornisce la sintassi per individuare variabili attraverso un indice. Una possibile traduzione dell’algoritmo appena dato, ma espresso in codice Java, che, per ora, assumiamo sia parte del metodo main, diventa:

```
1 public static void main(String[] args) {
2 int[] a = {15, 2, -3, 16, 9};
3 int tmp;
4 int i = 0;
5 while (i < a.length - 1) {
6 if (a[i] > a[i+1]) {
7 tmp = a[i]; // scambio dei valori in a[i] e a[i + 1]
8 a[i] = a[i + 1];
9 a[i + 1] = tmp;
10 }
11 i = i + 1;
12 }
13 }
```

La corrispondenza tra algoritmo e programma è quasi uno-a-uno. Gli elementi discordanti stanno nella dichiarazione dell’*array*  $a$ :

```
int[] a = {15, 2, -3, 16, 9};
```

e nella sostituzione dell'espressione:

```
a.length
```

al valore esplicito 5.

Per comprendere a fondo quel che succede occorre illustrare la gestione della memoria a fronte dell'introduzione degli *array*.

## 6.2 Array e gestione della memoria: *Heap*

L'introduzione della struttura dati *array* richiede una estensione del modello di memoria sviluppato sinora. A *static memory*, *frame stack* e *operand stack* occorre ora affiancare la zona di memoria identificata come *heap*.

In generale, la *heap* è deputata a contenere strutture dati la cui dimensione non è detto possa essere prevista sin dalla fase di compilazione.

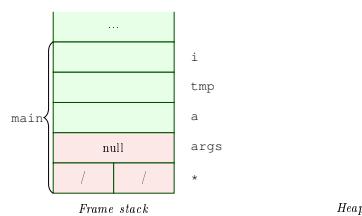
La motivazione appena addotta per giustificare l'introduzione della *heap* non sarà immediatamente evidente dai primi esempi.

Riprendiamo il programma precedente:

```
1 public static void main(String[] args) {
2 int[] a = {15, 2, -3, 16, 9};
3 int tmp;
4 int i = 0;
5 while (i < a.length - 1) {
6 if (a[i] > a[i+1]) {
7 tmp = a[i]; // scambio dei valori in a[i] e a[i + 1]
8 a[i] = a[i + 1];
9 a[i + 1] = tmp;
10 }
11 i = i + 1;
12 }
13 }
```

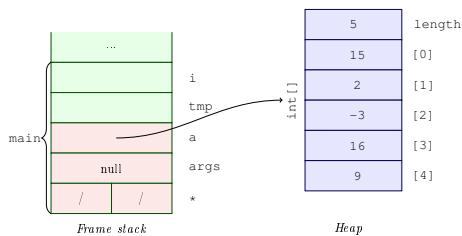
ed interpretiamolo.

Prima dell'interpretazione di riga 2, tranne che per un aspetto, la memoria è organizzata come atteso:



Il *frame* di *main* alloca spazio per *args*, *i* ed *a*, quest'ultima considerata alla stregua delle altre variabili. La novità sta nella comparsa della zona di memoria *heap*, per ora vuota.

Al termine dell'interpretazione della riga 2, la memoria si trova nella seguente situazione:

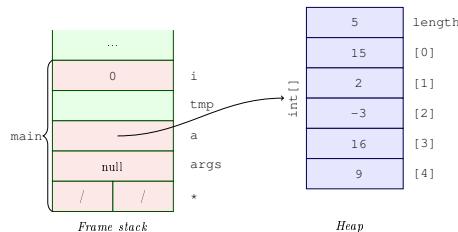


Nella memoria *heap* è comparsa la rappresentazione dell'*array* *a*:

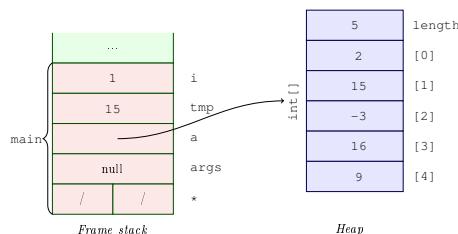
- all'intera struttura possiamo associare il tipo *int []* per ricordare che essa è un array i cui elementi contengono valori di tipo *int*.
- Il primo campo di nome *length* indica il numero di elementi indicizzabili. Siccome esso contiene il valore 5, per convenzione, gli indici possono variare tra 0 e 4 estremi inclusi.
- L'elemento di indice 0 è identificato con l'etichetta [0], quello di indice 1 con etichetta [1] e così via.

Il contenuto della cella `a` nel *frame* di `main` è un *riferimento* adatto a recuperare ogni informazione utile nella struttura di tipo `int[]` appena allocata. Sinonimi di “riferimento” sono “*indirizzo*” e “*puntatore*”. È usuale usare anche la parola “*reference*”, al posto di “riferimento”.

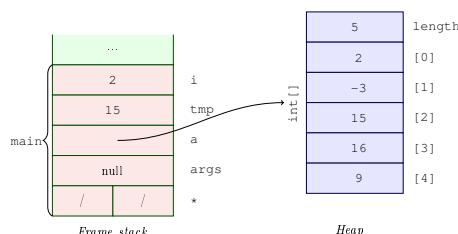
Al termine dell’istruzione 4 la situazione evolve come ci si può attendere, inizializzando la variabile `i`, ma non `tmp`:



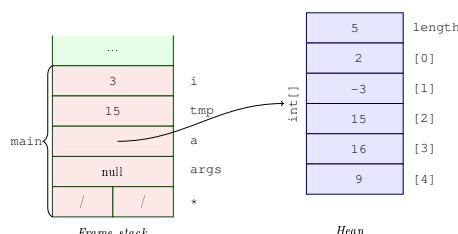
L’interpretazione del predicato argomento del costrutto iterativo `while` produce il valore `true` siccome `i == 0 < a.length - 1 == 4`. Segue l’interpretazione dell’espressione `a[0] > a[1]` che è vera perché `a[0] == 15` e `a[1] == 2`. L’interpretazione delle istruzioni 8, ..., 11 portano la memoria nella seguente situazione:



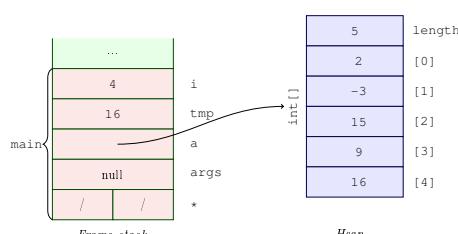
A questo punto valutiamo `1 == i < a.length - 1 == 4` argomento del costrutto iterativo. Siccome l’espressione produce `true` reinterpretiamo l’espressione `a[1] > a[2]` argomento della selezione che restituisce `true`. L’interpretazione delle istruzioni 8, ..., 11 portano la memoria nella seguente situazione:



Ricominciamo da `2 == i < a.length - 1 == 4`. Il valore è `true`. Possiamo valutare `a[2] > a[3]`, ottenendo `false`. L’unica istruzione eseguibile è la 11, ottenendo:



Ricominciamo da `3 == i < a.length - 1 == 4` che vale `true`. Possiamo valutare `a[3] > a[4]`, ottenendo `false`. Le istruzioni 8, ..., 11 restituiscono:



Ricominciando da `4 == i < a.length - 1 == 4` otteniamo `false`. L’iterazione termina proprio quando il massimo valore in `a` occupa la cella con indice maggiore.

**Esercizio 42 (Alternativa per l'emersione del massimo)** Definire `emersioneMax(int[])` che, applicato ad un parametro attuale `a` di tipo `int[]`, lo modifichi in modo che, al termine del metodo, il valore massimo in `a` sia in posizione `a[a.length - 1]`.

Nella progettazione del metodo `emersioneMax(int[])` sottostare al seguente vincolo: l'emersione non deve procedere scambiando, passo dopo passo, i valori di due celle contigue in `a`.

Possibili suggerimenti alternativi:

1. visitare `a` con lo scopo di ricordare l'indice del valore massimo. Quindi memorizzare il massimo nell'ultimo elemento di `a`, senza perdere quest'ultimo.
2. visitare `a` e, man mano che si incontra un massimo, metterlo in ultima posizione, senza perdere il valore in essa contenuto.

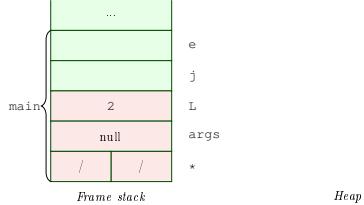
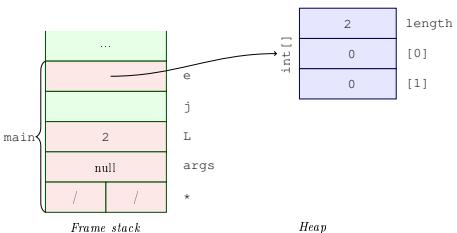
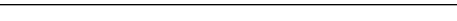
## 6.3 Creazione di *array*, inizializzazione ed egualianza

Creazione e stampa di *array*, anche col costrutto ciclico `for`, e modello di memoria associato.

```

1 public static void main(String[] args) {
2 final int L = 2;
3 int j;
4 int[] e;
5 e = new int[L];
6
7 for (j = 0; j < L; j++) {
8 e[j] = SIn.readInt();
9 }
10
11 for (j = 0; j < e.length; j++)
12 System.out.print(e[j]);
13 }
```

Segue l'interpretazione di blocchi di righe:

| Codice interpretato                           | Situazione della memoria                                                                                                                                                         |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> final int L = 2; int j; int[] e;</pre>  | Né <code>j</code> , né <code>e</code> sono inizializzate. Per essi è solo allocato spazio.   |
| <code>e = new int[L];</code>                  | In <code>e</code> c'è il <i>reference</i> ad una struttura per il tipo <code>int[]</code> .  |
| <pre> j = 0; e[j] = SIn.readInt(); j++;</pre> | Supponendo di inserire il valore <code>-1</code> , tramite lo standard input.                |

|                                       |                                                                |
|---------------------------------------|----------------------------------------------------------------|
|                                       | <p>Frame stack</p> <p>Heap</p>                                 |
| <pre>e[j] = SIn.readInt(); j++;</pre> | Supponendo di inserire il valore 1, tramite lo standard input. |
|                                       | <p>Frame stack</p> <p>Heap</p>                                 |

Quando  $j == 2$  si procede all'interpretazione delle istruzioni alle linee 11 e 12, il cui effetto è di reinizializzare  $j$  in modo da percorrere tutti gli elementi di  $a$ . Alla linea 13, la situazione della memoria è identica a quella della linea 10.

**Esercizio 43 (Correttezza parziale di egualianza (intensionale) tra array)** Assumiamo che i due array  $a$  e  $b$  abbiano lo stesso numero di elementi. Dimostrare che la correttezza parziale del seguente codice:

```

1 boolean uguali = true;
2 i = 0;
3 while (i < a.length - 1 && uguali) {
4 uguali = a[i] == b[i];
5 if (uguali) {
6 i = i + 1;
7 }
8 }
```

in cui, al termine dell'iterazione, ci si aspetta che  $uguali == true$  se  $a$  e  $b$  coincidono indice per indice. Altrimenti, ovvero se esiste un indice per cui gli elementi di  $a$  e  $b$  differiscono, deve essere che  $uguali == false$ .

La seguente soluzione possibile usa il predicato per ogni  $k$ , se  $0 \leq k < i$  allora  $a[k] == b[k]$  come invarianto:

```

1 boolean uguali = true;
2 i = 0;
3 // per ogni k, se 0 <= k < i allora a[k]==b[k] e' vero perche' la premessa e' vacua.
4 while (i < a.length && uguali) {
5 // per ogni k, se 0 <= k < i allora a[k]==b[k] vero per ipotesi (****)
6 uguali = a[i] == b[i];
7 if (uguali) {
8 // (per ogni k, se 0 <= k < i allora a[k]==b[k]) e a[i]==b[i]
9 // implica che
10 // per ogni k, se 0 <= k < i+1 allora a[k]==b[k]
11 i = i + 1;
12 // per ogni k, se 0 <= k < i allora a[k]==b[k]
13 // che coincide con (****)
14 }
15 }
16 // Supponiamo uguali == true. In tal caso necessariamente i == a.length.
17 // Quindi per ogni k, se 0 <= k < a.length allora a[k]==b[k] e' vero e
18 // questo significa a[0] == b[0], ... a[a.length-1] == b[a.length-1].
19 // Supponiamo uguali == false. In tal caso
20 // per ogni k, se 0 <= k < i allora a[k]==b[k] e' vero, ma a[i]!=b[i], quindi
21 // a e b sono diversi.
```

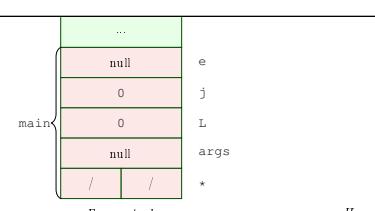
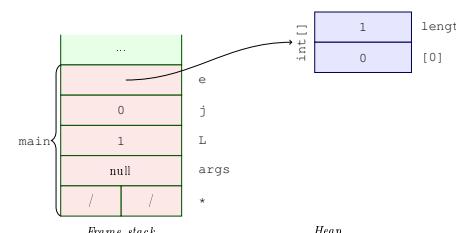
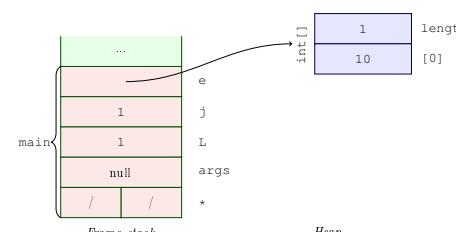
### 6.3.1 Perché gli array sono nella heap?

```

1 public static void main(String[] args) {
2 final int L;
3 int[] e;
4 int j;
5 L = SIn.readInt();
6 e = new int[L];
7 for (j = 0; j < e.length; j++) {
8 e[j] = SIn.readInt();
9 }
}

```

Segue l'interpretazione di blocchi di righe:

| Codice interpretato                             | Situazione della memoria                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> final int L; int[] e; int j; </pre>       |  <p>Il valore di <code>L</code> è quello di default. Non è ancora stato usato per definire il numero di elementi in <code>e</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                  |
| <pre> L = SIn.readInt(); e = new int[L]; </pre> |  <p>Il valore di <code>L</code> è noto solo dopo la lettura del suo valore tramite la tastiera, quindi ignoto durante la compilazione. Non è, quindi, possibile stabilire quanto spazio riservare ad <code>e</code> nel <i>frame</i> del <code>main</code>. Conseguentemente, l'allocazione di <code>e</code> avviene nello <i>heap</i>. Al contrario, per un qualsiasi dato di tipo primitivo lo spazio necessario è noto durante la compilazione ed è possibile riservare nel <i>frame</i> lo spazio necessario.</p> |
| <pre> e[j] = SIn.readInt(); j++; </pre>         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

## 6.4 Eguaglianza tra *array* e *aliasing*

Per le strutture dati non primitive come gli *array* è necessaria una definizione esplicita di eguaglianza. Il motivo è evidente dall'interpretazione dei metodi nella classe seguente:

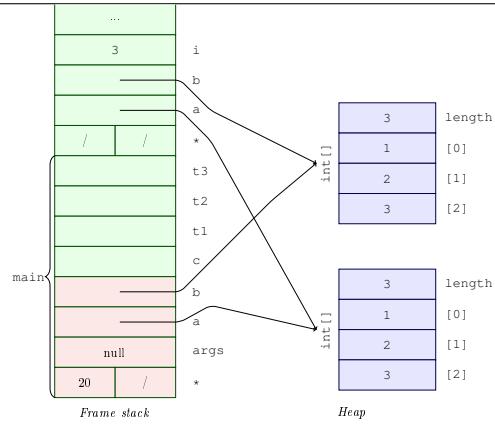
```

1 public class ArrayCopieCloni {
2 public static void arrayClone(int[] a, int[] b) {
3 int i;
4 for (i = 0; i < a.length; i++)
5 b[i] = a[i];
6 }
7
8 public static int[] arrayClone(int[] b) {
9
10 int[] a = new int[b.length];
11 arrayClone(b, a);
12 return a;
13 }
14
15 public static void main(String[] args) {
16
17 int[] a = { 1, 2, 3 };
18 int[] b = new int[a.length];
19 int[] c;
20 arrayClone(a, b);
21 c = arrayClone(b);
22 boolean t1 = (a == b);
23 boolean t2 = (c == b);
24 boolean t3 = (a == c);
25 }
26 }
```

Il codice è organizzato anche per evidenziare il fenomeno dell'*aliasing*. Con “*aliasing*” si intende l'esistenza di almeno due variabili *a* e *b*, anche in *frame* distinti, tali che il risultato di un'azione su una delle due variabili equivale all'eseguire la stessa azione sull'altra. Il fenomeno esiste proprio perché *a* e *b* possono essere *reference* alla stessa struttura che si trova nello *heap*.

La seguente interpretazione della classe *ArrayCopieCloni* evidenzia sia l'inefficacia dell'operatore `==` nello stabilire l'eguaglianza tra più *array*, sia l'*aliasing*:

| Codice interpretato                                                  | Situazione della memoria                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> int[] a = {1,2,3}; int[] b = new int[a.length]; int[] c;</pre> | <p>The diagram illustrates the memory state after the execution of the provided Java code. On the left, the <i>Frame stack</i> is shown with the <i>main</i> frame containing local variables <i>t3</i>, <i>t2</i>, <i>t1</i>, <i>c</i>, <i>b</i>, <i>a</i>, <i>null</i>, <i>args</i>, and the stack pointer <i>*/</i>. The <i>Heap</i> contains two <i>int[]</i> arrays. The first array, located at address 20, has elements [1, 2, 3]. The second array, located at address 30, also has elements [1, 2, 3]. Arrows from the <i>Frame stack</i> indicate that <i>b</i> points to the array at 20 and <i>a</i> points to the array at 30.</p> |
| arrayClone(a, b);                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |



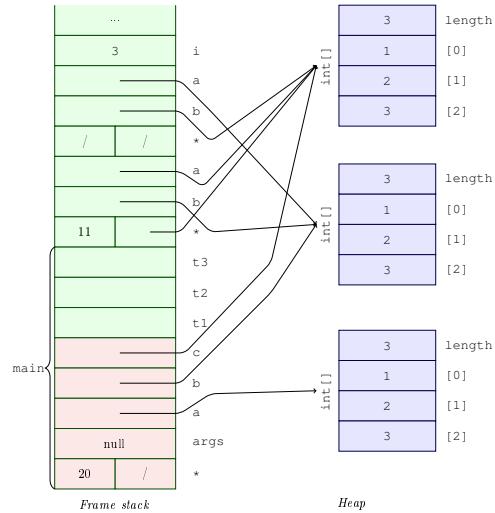
In cima al *frame stack*, seppure in veste di spazio riutilizzabile, esiste intatta la struttura del *frame* per `arrayClone(int[], int[])`.

Inoltre, è evidente che `a` e `b` nel *frame* di `arrayClone` sono *alias* delle omonime variabili del `main`. Ovvero, modificando le prime si modificano le seconde. Al termine di `arrayClone`, la variabile `b` in `main` punta ad una istanza di *array* che mantiene le modifiche apportate da `arrayClone` alla propria istanza di `b`.

---

```
c = arrayClone(b);
```

---

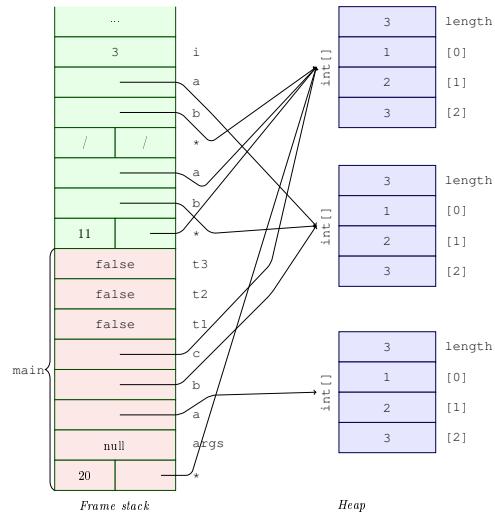


In cima al *frame stack*, partendo dall'alto, si possono vedere le strutture dei due *frame* relativi a `arrayClone(in[], int[])` e `arrayClone(in[])`.

---

```
boolean t1 = (a == b);
boolean t2 = (c == b);
boolean t3 = (a == c);
```

---



È evidente che l'operatore `==` non sia sufficiente ad esprimere il concetto di egualianza (intensionale) intuitivo e ragionevole tra *array* le cui celle abbiano tipo primitivo `int`. Anche se `a`, `b` e `c` hanno identica lunghezza e contengono elementi identici in identica posizione, il valore `t1`, `t2` e `t3` è sempre `false`.

#### 6.4.1 Classe Array

Java fornisce la classe `Array`, parte del *package* `java.utils` tale che `Array.equals`, applicato a due *array* dello stesso tipo ne verifica l'egualianza strutturale che, almeno per *array* che contengano elementi di tipo base, è quella "ovvia": ovvero, due *array*, ad esempio di tipo `int[]`, sono uguali se di identica lunghezza, e se coincidenti, elemento per elemento.

**Esercizio 44 (Metodo equivalente ad `Array.equals`.)** Data la seguente classe:

```

import java.util.Arrays;
2
public class ArrayEqualianza {
4
 public static boolean arrayIntEquals(int[] a, int[] b) {
6
 boolean forseUguali = (a == null && b == null) // entrambi vuoti
8
 || (a != null && b != null && a.length == b.length); // non vuoti, lunghi uguali
 int i;
10 if (a != null && b != null && forseUguali) {
 for (i = 0; i < b.length && forseUguali; i++)
 forseUguali = (a[i] == b[i]);
 }
14 return forseUguali;
}
16
public static void main(String[] args) {
18
 int[] a = {1};
20 int[] b = {1};
21 boolean test1 = arrayIntEquals(a, b);
22 boolean test2 = Arrays.equals(a, b);
}
24 }
```

- compilare ed interpretare, sperimentando la variazione dei valori in `test1` e `test2` al variare di `int[] a = {1}; int[] b = {1};`, come segue:
  - `int[] a = null;`
  - `int[] b = null;`
  - oppure `int[] a = {1};`
  - `int[] b = null;`
  - oppure `int[] a = {1};`
  - `int[] b = {1, 2};`

```
- oppure int[] a = {1};
 int[] b = {2}; .
```

- Disegnare l'evoluzione della memoria, scegliendo tra una o più delle combinazioni appena date per inizializzare a e b.
- Ovviamente, è possibile scrivere il metodo `arrayIntEquals(int[], int[])` ricorsivamente. Di seguito, un paio di versioni possibili:

```
public static boolean arrayIntEqualsControvariante(int[] a, int[] b) {
 boolean forseUguali = (a == null && b == null) // entrambi vuoti
 || (a != null && b != null && a.length == b.length); // non vuoti, lunghi uguali
 if (a != null && b != null && forseUguali)
 return arrayIntEqualsControvariante(a, b, 0);
 return forseUguali;
}

private static boolean arrayIntEqualsControvariante(int[] a, int[] b, int i) {
 if (i < a.length)
 return a[i] == b[i] && arrayIntEqualsControvariante(a, b, i + 1);
 else
 return true;
}

public static boolean arrayIntEqualsCovariante(int[] a, int[] b) {
 boolean forseUguali = (a == null && b == null) // entrambi vuoti
 || (a != null && b != null && a.length == b.length); // non vuoti, lunghi uguali
 if (a != null && b != null && forseUguali)
 return arrayIntEqualsCovariante(a, b, a.length - 1);
 return forseUguali;
}

private static boolean arrayIntEqualsCovariante(int[] a, int[] b, int i) {
 if (i < a.length)
 return a[i] == b[i] && arrayIntEqualsCovariante(a, b, i - 1);
 else
 return true;
}
```

Sui due metodi ricorsivi, è interessante osservare le conseguenze di sostituire la linea di codice

```
1 a[i] == b[i] && arrayIntEqualsControvariante(a, b, i + 1);
```

con

```
1 arrayIntEqualsControvariante(a, b, i + 1) && a[i] == b[i];
```

oppure

```
1 a[i] == b[i] && arrayIntEqualsCovariante(a, b, i - 1);
```

con

```
1 arrayIntEqualsCovariante(a, b, i - 1) && a[i] == b[i];
```

## 6.5 Operazioni di base su *array*

### 6.5.1 Ricerca lineare

[RicercaLineare.java](#) per la quale è naturale definire il costo, contando il numero di confronti eseguiti prima di individuare l'elemento, se esiste. Nel caso peggiore occorre visitare tutti gli elementi, ed il costo è pari a `a.length - 1` confronti. Quindi, in generale, il costo della ricerca lineare cresce linearmente con la dimensione dell'array cui è applicata.

### 6.5.2 Filtri

[Filtri.java](#), [FiltriTest.java](#) forniscono alcuni esempi di proiezioni di elementi, che godono di specifiche proprietà, da un *array*.

**Esercizio 45 (Possibile filtro)** Scrivere una classe con un metodo ricorsivo `void pariPrimaDeiDispari(int[] a)` che riorganizzi gli elementi di *a* in modo che quelli di valore pari precedano gli elementi di valore dispari. L'ordine relativo finale dei valori pari e dispari è irrilevante.

(La soluzione è in [BatteriaEserciziRecLaboratorio.java](#)) ■

### 6.5.3 Inserimenti e cancellazioni

[Inserimento.java](#), [InserimentoTest.java](#) forniscono due soluzioni iterative per inserire il valore *e* in posizione *p* di un array *a*.

La correttezza parziale di entrambe le soluzioni è descritta in [Invariante dell'inserimento di un elemento](#).

La seconda soluzione è più efficiente della prima, se contiamo il numero di assegnazioni eseguite. Nel primo caso si eseguono sia le  $a.length$  assegnazioni per copiare tutti gli elementi di *a* in *b*, sia le  $a.length - (p + 1)$  assegnazioni per spostare verso destra gli elementi, prima dell'inserimento. Il totale delle assegnazioni diventa  $a.length + a.length - (p + 1) + 1 = 2 * a.length - p$ .

Nel secondo caso si eseguono *p* assegnazioni degli elementi a sinistra di *p* cui aggiungere le  $a.length - (p + 1)$  assegnazioni degli elementi alla sua destra, più l'inserimento. Il totale diventa  $p + a.length - (p + 1) + 1 = a.length$ .

In entrambi i casi, il numero di assegnazioni cresce linearmente al crescere della lunghezza dell'array. Quindi, al crescere della lunghezza, la differenza in efficienza perde rilevanza.

[Cancellazione.java](#) e [CancellazioneTest.java](#) forniscono due soluzioni per la cancellazione dell'elemento di indice *p* in un array *a*.

La correttezza parziale della seconda soluzione è descritta in [Invariante dell'eliminazione di un elemento](#).

L'efficienza delle due soluzioni, misurata contando il numero di assegnazioni eseguite differisce di un valore costante.

Nel primo caso, si esegue una prima assegnazione, seguita da  $a.length - 2$  assegnazioni che copiano tutti gli elementi di *a*, tranne l'ultimo, in *b*.

Nel secondo caso, si eseguono solo  $a.length - 2$  assegnazioni tutti gli elementi di *a*, tranne quello di posizione *p* in *b*.

In entrambi i casi, il numero di assegnazioni cresce linearmente al crescere della lunghezza dell'*array*.

## 6.6 Array e visite dicotomiche o parallelizzabili

Nella Sottosezione 5.4.4 o in alcuni dei punti dell'Esercizio 37 si descrivono funzioni la cui dimostrazione di correttezza parziale richiede l'uso del principio di induzione forte, o completo. La necessità nasce dal fatto che i richiami ricorsivi sono applicati ad argomenti il cui valore non è semplicemente incrementato e decrementato di una unità.

### 6.6.1 Schema dicotomico generale

Nel definire algoritmi su *array* possiamo applicare lo stesso principio:

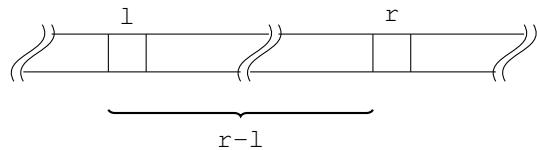
Sia data una certa proprietà da risolvere su un *array* *a*. Ad esempio, vogliamo contare il numero di elementi in *a* con una certa proprietà, o vogliamo semplicemente verificare l'esistenza di un certo elemento in *a*.

Per risolvere l'intero problema, possiamo immaginare di saperlo risolvere induttivamente su due porzioni di *a* di lunghezza essenzialmente identica, ovvero su porzioni nettamente più corte dell'*array* iniziale. Una volta ottenuta la soluzione ai due problemi "più piccoli", componiamo le soluzioni per ottenere quella finale.

Il passo base consiste nel saper risolvere il problema quando l'*array* disponibile abbia un solo elemento.

Il processo è realizzabile in almeno due modi equivalenti. Essi si differenziano nel modo in cui individuano gli estremi degli intervalli in cui spezzare la porzione di *array* su cui si opera inizialmente.

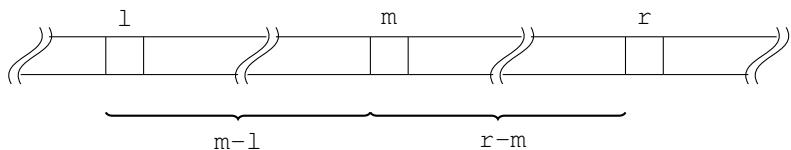
**Primo approccio.** L'ipotesi di partenza è di dover risolvere il problema sulla porzione  $a[1] \dots a[r-1]$  d'*array* *a*, come segue:



Ovvero è fondamentale immaginare che l'invariante da mantenere si possa esprimere come:

$l$  è l'indice del primo elemento effettivo dell'intervallo su cui operare, mentre  $r$  è l'indice successivo all'ultimo elemento nell'intervallo.

Il numero di celle compreso tra i due estremi è  $r-1$ , assumendo  $r$  maggiore od uguale ad 1. Calcolando la media tra i due estremi, ovvero  $m = (l+r)/2$ , ed approssimandola all'intero inferiore nel caso  $l+r$  sia dispari, otteniamo:



in cui, rispettando l'invariante,  $m-l$  è il numero di celle nell'intervallo  $a[1] \dots a[m-1]$  e  $r-m$  quello delle celle nell'intervallo  $a[m] \dots a[r-1]$ . Ovvero, l'intervallo iniziale di  $r-1$  celle è diviso in due parti consecutive più piccole, avendo  $r-m + (m-l) = r-m+m-1 = r-1$ .

Il caso limite si ha quando lo scarto tra i due valori è minimo, ovvero quando  $r-1$  vale 1. In tal caso l'intervallo è costituito da una singola cella sulla quale dobbiamo poter risolvere il problema senza ulteriori scomposizioni.

Un algoritmo che esprima tale schema generale può avere la seguente struttura:

```

1 public static <tipo-opportuno> schemaDicotomico(int[] a) {
2 if(a != null)
3 return schemaDicotomico(a, 0, a.length);
4 else
5 return ... // eventuale valore di tipo opportuno
6 }
7
8 private static <tipo-opportuno> schemaDicotomico(int[] a, int l, int r) {
9 if (l < r)
10 if (r - l == 1)
11 return ... // eventuale risultato nel caso base
12 else {
13 int m = (l+r)/2;
14 return composizioneOppertaDe(schemaDicotomico(a, l, m),schemaDicotomico(a, m, r));
15 }
16 else
17 return ... // eventuale risultato nel caso lo schema induttivo non sia piu' applicabile
}
```

**Esempio 10 (Esempi di algoritmi dicotomici su array (Primo metodo))** [DicotomiaSuArray.java](#) fornisce un paio di metodi dicotomici. Applicati ad un *array*, il primo metodo restituisce true nel caso uno specifico valore  $k$  esista nell'*array*; il secondo metodo conta il numero di occorrenze di  $k$  in *a*.

Ad esempio, riguardo al metodo conteggio(int[], int) della classe [DicotomiaSuArray.java](#) forniamo la dimostrazione di correttezza parziale che sfrutta il principio di induzione forte. Con esso dimostriamo il predicato:

Per ogni *a* ed ogni  $k$ , il metodo conteggio(*a*,  $k$ ,  $l$ ,  $r$ ) restituisce il numero delle occorrenze di  $k$  in  $a[1] \dots a[r-1]$ .

procedendo per induzione forte sul valore  $r-1$ .

Consideriamo le due chiamate ricorsive

```

conteggio(a, k, l, m)
conteggio(a, k, m, r)
```

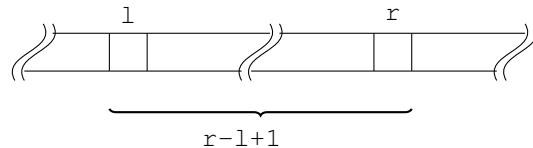
in cui  $m == (l+r)/2$ .

- Supponiamo che  $l < r$ . Allora  $l < m < r$ . Quindi  $r-1 > r-m$  e  $m-l < r-m$ . Valendo l'ipotesi di induzione forte otteniamo che  $\text{conteggio}(a, k, l, m)$  fornisce il numero di occorrenze di  $k$  in  $a[1] \dots a[m-1]$  e  $\text{conteggio}(a, k, m, r)$  fornisce il numero di occorrenze di  $k$  in  $a[m] \dots a[r-1]$ . La somma dei due valori non può che dare la somma delle occorrenze di  $k$  nell'intervallo  $a[1] \dots a[r-1]$ , ottenuto dai due intervalli contigui in *a*.

- Supponiamo  $r-l==1$ , ovvero il numero di celle nell'intervallo  $a[1] \dots a[r-1]$  è pari ad 1. Necessariamente contiamo 1 se  $k$  coincide col valore in  $a[1]$ .

■

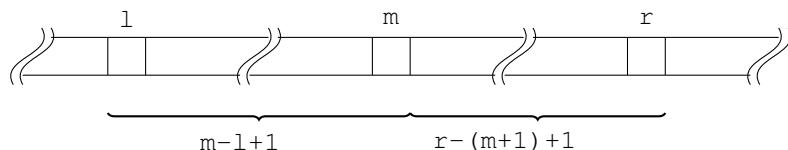
**Secondo approccio.** L'ipotesi di partenza è di dover risolvere il problema sulla porzione  $a[1] \dots a[r]$  d'array  $a$ , come segue:



Ovvero è fondamentale immaginare che l'invariante da mantenere si possa esprimere come:

$l$  è l'indice del primo elemento effettivo dell'intervallo su cui operare, mentre  $r$  è l'indice dell'ultimo elemento nell'intervallo.

Il numero di celle compreso tra i due estremi è  $r-l+1$ , con  $r$  maggiore o uguale 1. Calcolando la media tra i due estremi, ovvero  $m = (l+r)/2$ , ed approssimandola all'intero inferiore nel caso  $l+r$  sia dispari, otteniamo:



in cui, rispettando l'invariante,  $m-l+1$  è il numero di celle nell'intervallo  $a[1] \dots a[m]$  e  $r-(m+1)+1$  quello delle celle nell'intervallo  $a[m+1] \dots a[r]$ . Ovvero, l'intervallo iniziale di  $r-l+1$  celle viene diviso in due parti consecutive più piccole, avendo  $(r-(m+1)+1)+(m-l+1) = r-m-1+m-1+1 = r-l+1$ .

Il caso limite si ha quando rimane un singolo elemento, ovvero  $r==l$ . Su quella singola cella di  $a$  occorre poter risolvere il problema senza ulteriori scomposizioni.

Lo schema generale di questo approccio diventa il seguente:

```

public static <tipo-opportuno> schemaDicotomico(int[] a) {
 2 if(a != null)
 return schemaDicotomico(a, 0, a.length - 1);
 4 else
 return ... // eventuale valore di tipo opportuno
 6 }

8 private static <tipo-opportuno> schemaDicotomico(int[] a, int l, int r) {
 10 if (l <= r)
 11 if (r == l)
 12 return ... // eventuale risultato nel caso base
 13 else {
 14 int m = (l+r)/2;
 15 return composizioneOpportunaDe(schemaDicotomico(a, l, m),schemaDicotomico(a, m + 1, r));
 16 }
 18 else
 19 return ... // eventuale risultato nel caso lo schema induttivo non sia piu' applicabile
}

```

**Esercizio 46 (Esercizi dicotomici)** Scrivere i seguenti metodi, usando definizioni ricorsive in modo da suddividere dicotomicamente l'array cui sono applicati:

1. int minimo(int[] a) che restituisca il valore minimo in a.
2. int contaMaggioriDiK(int[] a, int k) che restituisca il numero dei valori di a che siano maggiori di k.
3. boolean esistonoMaggioriDiK(int[] a, int k) che restituisca il true nel caso esista almeno un elemento in a che sia maggiore di k.
4. boolean tuttiMaggioriDiK(int[] a, int k) che restituisca il true nel caso tutti gli elementi in a siano maggiori di k.
5. int tuttiZeroUno(int[] a) che, applicato ad a, array i cui valori siano presi dall'insieme {0,1} (i) 1 se a contiene solo occorrenze di 1, (ii) 0 se a contiene solo occorrenze di 0, (iii) -1 altrimenti.

6. boolean crescente(int[] a) che restituisca true se a è ordinata in maniera debolmente crescente.

[EserciziDicotomici.java](#) contiene possibili soluzioni agli esercizi proposti, seguendo il secondo approccio, ma ristrutturando lo schema generale per sfruttare il fatto che l'estremo superiore *r* dell'intervallo non possa mai scendere al di sotto dell'estremo inferiore *l*. Un utile ulteriore esercizio è riformulare ogni soluzione in accordo con il primo approccio, eventualmente ristrutturando lo schema generale per sfruttare il fatto che la differenza tra l'estremo superiore *r* dell'intervallo e l'estremo inferiore *l* non possa mai scendere al di sotto del valore 1. ■

## 6.7 Matrici bidimensionali

È possibile annidare *array* in *array* col fine di costruire strutture (almeno) bidimensionali. Osserviamo il seguente frammento di codice:

```

1 int[][] m = null;
2 m = new int[4][];
3 int[] r0 = { 0, 1, 2 };
4 int[] r1 = { 3, 4 };
5 int[] r2 = { 5 };
6 int[] r3 = { 6, 7, 8 };
7 m[0] = r0;
8 m[1] = r1;
9 m[2] = r2;
10 m[3] = r3;

```

Alcune osservazioni:

- Riga 1: *m* assume valore *null*, ma la dichiarazione di tipo esprime il fatto che *m* potrà puntare ad un *array* le cui componenti potranno a loro volta essere *array* di interi.
- Riga 2: ad *m* è assegnato un *reference* ad una struttura nella memoria *heap*. Tale struttura ha tipo *int[][]* e contiene quattro elementi che, come al solito, possiamo indicare come *m[0]*, *m[1]*, *m[2]*, *m[3]*. Ciascuno degli elementi *m[i]* ha tipo *int[]*. Quindi è adatto ad assumere il valore di un *reference* ad un *array* di tipo *int[]*.
- Righe dalla 3 alla 6: nello *heap* costruiscono, ciascuna un *array* di tipo *int[]*. È evidente che *r1*, *r2*, ... non hanno lunghezza identica.
- Righe dalla 7 in poi alla 10: ogni elemento *m[i]* assume il *reference* all'*array* di interi con medesimo indice. La matrice finale ottenuta in *m* è *ragged* perché con righe di diversa lunghezza.

[Matrici.java](#) è una raccolta di metodi il cui scopo è evidenziare come “muoversi” su matrici bidimensionali, anche *ragged*.

**Esercizio 47** • Scrivere un metodo *String toString(int[][] m)* che riversi l'intero contenuto della matrice in una stringa, strutturandolo in maniera ragionevole: gli elementi di *m* su righe diverse devono risultare su righe diverse anche nel risultato.

- Scrivere un metodo *int[] diagonalePrincipale(int[][] m)* che produca un *array* con tutti gli elementi della diagonale principale di *a*.
- Scrivere un metodo *int[] diagonaleSecondaria(int[][] m)* che produca un *array* con tutti gli elementi della diagonale secondaria di *a*.
- Scrivere un metodo *boolean quadrata(int[][] m)* che restituisca true nel caso *m* sia quadrata, non *ragged*, false altrimenti. Una matrice bidimensionale è *ragged*, ovvero “sfrangiata” se ha righe di diversa lunghezza.
- Scrivere il metodo statico *int[][] per(int[][] a, int[][] b)* che date due matrici *a* e *b* ne produca una terza che rappresenti il prodotto matriciale di *a* e *b*.
- Sia dato un array *mD* che sta per “mono dimensionale”. Creare una matrice bidimensionale *bD* con un numero di colonne *nC* prefissato ed abbastanza righe nella quale copiare, riga per riga, tutte gli elementi di *mD*. Ad esempio, se *mD* è l'*array* {2, 4, 6, 7, 8, 1, 9} e *nC* è fissato al valore 3, allora *bD* è la matrice bidimensionale:

```

{ {2, 4, 6}
{7, 8, 1}
{9, 0, 0}}

```

nella quale le due occorrenze di 0 sono i valori di default assegnati alle celle di un array, se non altrimenti definite. Per creare `bD` è possibile immaginare tre metodi distinti. Ciascuno di essi scorrerà `mD` con un indice, diciamo `i` e `bD` con due indici, ad esempio `r` e `c`. I tre metodi si differenziano l'un l'altro per il modo in cui vengono aggiornati i valori di `r` e `c`, al variare di `i`. Nel primo metodo non si usa alcuna operazione modulare. Nel secondo si applicano operazioni modulari per aggiornare correttamente `r`. Nel terzo, le operazioni modulari sono usate per aggiornare i valori di entrambi `r` e `c`. Si sottolinea che le operazioni modulari permettono di eliminare l'uso del costrutto selezione `if-then-else`.

`MonoBidimensionale.java` contiene una possibile soluzione all'ultimo esercizio. Il punto dell'esercizio è rendersi conto che matrici bidimensionali possono essere "comprese" in array monodimensionali in cui occorre immaginare di raggruppare opportunamente sequenze di celle corrispondenti a righe della matrice di partenza.

`OperazioniSuMatrici.java` contiene un insieme di possibili soluzioni a problemi posti nei punti precedenti. ■

### 6.7.1 Problemi decisionali

Le matrici con almeno due dimensioni si prestano alla definizione di problemi decisionali la cui specifica è esprimibile combinando nei quattro modi disponibili le quantificazioni universale ed esistenziale, applicate ad un qualche predicato  $P$ . Nella loro forma più essenziale questi problemi si possono riassumere come segue:

Data una matrice `m`, scrivere un metodo che risponde `true` in ciascuno di questi casi:

1. Esiste una riga `m[i]`, nella quale esiste un elemento `m[i][j]` tale che  $P$ ?
2. Per ogni riga `m[i]`, esiste un elemento `m[i][j]` tale che  $P$ ?
3. Esiste una riga `m[i]`, tale che ogni elemento `m[i][j]` sia tale che  $P$ ?
4. Per ogni riga `m[i]`, ogni elemento `m[i][j]` è tale che  $P$ ?

Esistono diversi approcci per la soluzione. Di seguito, evidenziamo una strategia risolutiva che, con piccole variazioni, permette di risolvere ciascuno dei quattro casi.

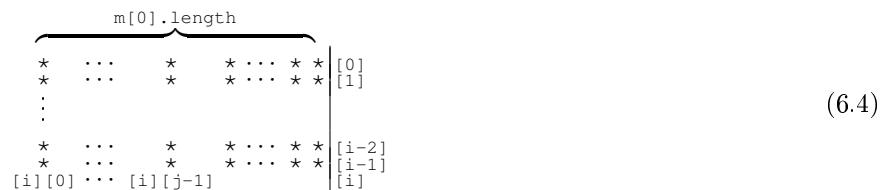
#### Esiste/PerOgni

Supponiamo di aver definito un metodo `p(int[][] m, int i, int j)` che restituisce un valore di tipo `boolean`, in funzione di una qualche proprietà goduta dall'elemento `m[i][j]` della matrice `m` non *ragged*.

Un metodo che dovrà restituire `true` se, e solo se, il predicato:

$$\exists i. (0 \leq i < m.length) \ \&\& \forall j. (0 \leq j < m[i].length \rightarrow p(m, i, j)) \quad (6.3)$$

è vero, dovrà necessariamente trovarsi nella seguente situazione intermedia:



nella quale ogni `*` indica un qualche valore numerico memorizzato nella corrispondente cella di `m`.

La configurazione si può leggere come segue:

- Nella riga `m[0]` esiste un qualche elemento `m[0][j]` che non soddisfa `p(m, 0, j)`. Nella riga `m[1]` esiste un qualche elemento `m[1][j]` che non soddisfa `p(m, 1, j)`. ... Nella riga `m[i-1]` esiste un qualche elemento `m[i-1][j]` che non soddisfa `p(m, i-1, j)`.

Se non fosse così, il metodo avrebbe già dovuto restituire `true` in corrispondenza della prima riga, ad esempio di indice `0 \leq k < i`, tale che `p(m, k, 0) \ \&\& \dots \&\& p(m, k, m[k].length - 1)` sia vero.

- Nella riga `m[i]` deve essere vero il predicato `p(m, i, 0) \ \&\& \dots \&\& p(m, i, j - 1)`.

Se non fosse così, il metodo dovrebbe essere già passato al controllo della riga successiva perché avrebbe trovato un qualche `0 \leq k \leq j - 1` per cui `p(m, i, k)` è `false`.

Conseguentemente alla lettura del significato della configurazione qui sopra esposta, è possibile stabilire come passare alla configurazione successiva in funzione del valore di `p(m, i, j)`. Occorre controllare la validità di ogni elemento nella riga `m[i]`. A tal fine serve un'iterazione come la seguente:

```

1 boolean ognielementoSoddisfaP = true;
2 int j = 0;
3 while (ognielementoSoddisfaP && j < m[i].length) {
4 ognielementoSoddisfaP = p(m,i,j);
5 j++;
6 } // j == m[i].length || !ognielementoSoddisfaP

```

Supponiamo che all'uscita dell'iterazione il predicato `!ognielementoSoddisfaP` sia vero. Siccome la riga `m[i]` non soddisfa il predicato  $P$ , rappresentato da `p`, su ogni suo elemento, allora occorre necessariamente scoprire se non sia la riga `m[i+1]` a soddisfare le richieste, ammesso che `m[i+1]` esista. Ne consegue che il frammento di codice appena dato va inserito in un'iterazione che dovrà essere eseguita per il numero necessario di volte. Un'approssimazione dell'iterazione necessaria è:

```

1 int i = 0;
2 while (i < m.length) {
3 boolean ognielementoSoddisfaP = true;
4 int j = 0;
5 while (ognielementoSoddisfaP && j < m[i].length) {
6 ognielementoSoddisfaP = p(m,i,j);
7 j++;
8 } // j == m[i].length || !p(m,i,j)
9 i++;
10 }

```

il cui significato si può leggere come segue:

Per ogni riga, si assume che ogni suo elemento soddisfa  $P$ , imponendo che `ognielementoSoddisfaP == true`. Non appena un elemento della riga su cui si sta valutando  $P$  ha un elemento per cui  $P$  è falso, allora occorre passare alla successiva.

Tuttavia, non appena si incontra una riga nella quale tutti gli elementi soddisfano  $P$  è inutile proseguire nella ricerca perché il predicato (6.3) è vero. Ovvero, il ciclo più esterno va interrotto. A tal fine è utile introdurre una nuova variabile:

```

1 boolean esisteRiga = false;
2 int i = 0;
3 while (!esisteRiga && i < m.length) {
4 boolean ognielementoSoddisfaP = true;
5 int j = 0;
6 while (ognielementoSoddisfaP && j < m[i].length) {
7 ognielementoSoddisfaP = p(m,i,j);
8 j++;
9 } // j == m[i].length || !p(m,i,j)
10 esisteRiga = ognielementoSoddisfaP;
11 i++;
12 }

```

Il significato del nuovo frammento di codice si può leggere come segue:

Si assume che non esistano righe in cui tutti gli elementi soddisfano  $P$ . Quindi, per ogni riga, si assume che ogni elemento soddisfa  $P$  e si cerca di verificare tale condizione.

Non appena un elemento della riga su cui si sta valutando  $P$  ha un elemento per cui  $P$  è falso, ovvero non appena `ognielementoSoddisfaP == false`, occorre passare alla successiva, ribadendo che `esisteRiga == false` grazie all'assegnazione `esisteRiga = ognielementoSoddisfaP`.

Al contrario, se ogni elemento della riga su cui si sta valutando  $P$  ha un elemento per cui  $P$  è vero, allora `ognielementoSoddisfaP == true`. In tal caso l'assegnazione `esisteRiga = ognielementoSoddisfaP` impone `esisteRiga == true` e l'iterazione più esterna *non* viene più ripetuta.

Il codice finale del metodo diventa quindi:

```

1 static boolean esistePerOgni (int[][] m) {
2 boolean esisteRiga = false;
3 int i = 0;
4 while (!esisteRiga && i < m.length) {
5 boolean ognielementoSoddisfaP = true;
6 int j = 0;
7 while (ognielementoSoddisfaP && j < m[i].length) {
8 ognielementoSoddisfaP = p(m,i,j);
9 j++;
10 } // j == m[i].length || !p(m,i,j)
11 esisteRiga = ognielementoSoddisfaP;
12 i++;
13 }

```

```

8 int j = 0;
9 while (ogniElementoSoddisfaP && j < m[i].length) {
10 ogniElementoSoddisfaP = p(m, i, j);
11 j++;
12 }
13 esisteRiga = ogniElementoSoddisfaP;
14 i++;
15 }
16 }
```

**Esempio 11** Una raccolta di esempi specifici di problemi decisionali Esiste/Esiste, Esiste/PerOgni, PerOgni/Esiste e PerOgni/PerOgni è nella classe [ProblemiDecisionaliSuMatrici.java](#).

Assumiamo sia dato il metodo `P` che, preso un intero  $v$  come parametro attuale, restituisce `true` se  $v$  è pari.

Assumiamo di applicare i metodi in [ProblemiDecisionaliSuMatrici.java](#) a sole istanze `m` di matrici bidimensionali di interi non *ragged*. I metodi risolvono i seguenti problemi:

- Esiste un indice  $r$  di riga per cui esiste un indice  $c$  di colonna tale che  $P(m[r][c])$  è vero?
- Esiste un indice  $r$  di riga tale che, per ogni indice di colonna  $c$ , l'elemento  $P(m[r][c])$  è vero?
- Per ogni indice di riga  $r$ , esiste un indice  $c$  di colonna tale che  $P(m[r][c])$ ?
- Per ogni indice di riga  $r$  e per ogni indice di colonna  $c$ , l'elemento  $P(m[r][c])$  è vero? ■

**Esercizio 48** 1. Modificare i metodi della classe [ProblemiDecisionaliSuMatrici.java](#) affinché possano essere applicati ad una qualsiasi matrice bidimensionale di interi.

2. Scrivere almeno tre *test* significativi per ciascuno dei metodi scritti al precedente punto. ■

### 6.7.2 Ricerca dicotomica su *array* ordinati

Lo schema generale di visita dicotomica della sezione precedente può essere specializzato se l'*array* su cui si opera è ordinato.

L'esempio classico di specializzazione è rappresentato dalla ricerca dell'esistenza di un elemento. Sapendo che l'*array* è ordinato, non è necessario cercare il valore dato nelle parti di *array* in cui è sicuro che esso non possa esistere, informazione data proprio dal fatto che esiste un ordine tra gli elementi dell'*array*.

[RicercaDicotomicaRec.java](#) e [RicercaDicotomicaIter.java](#) realizzano un algoritmo ricorsivo ed iterativo, rispettivamente, per la ricerca dicotomica in un *array* ordinato, seguendo il primo approccio.

## 6.8 Ordinamenti ed operazioni su *array* ordinati (NON è parte del programma didattico)

La Sezione 6.3. de [\[SM14\]](#), Capitolo 6] può essere di riferimento.

Il problema computazionale dell'ordinamento è di fondamentale importanza per le conseguenze positive che può aver il lavorare su strutture dati i cui valori siano ordinati. Ci concentreremo sull'ordinamento di *array* con valori numerici interi. In generale, ordinare una sequenza di elementi in un dominio  $\mathbb{D}$  per il quale esista una relazione d'ordine  $\mathcal{R}$  equivale a calcolare una permutazione  $P : \mathbb{D} \rightarrow \mathbb{D}$  tale che, per ogni (multi-insieme)  $X \in \mathbb{D}$ , si abbia  $P(X) = Y$  tale che (il multi-insieme)  $Y$  sia  $\langle y_1, \dots, y_n \rangle$  con tutti e soli gli elementi di  $X$  tale che  $\forall i. i \in \{1, \dots, n\} \Rightarrow y_i \mathcal{R} y_{i+1}$ . Ad esempio, se  $\mathbb{D}$  coincide con l'insieme dei numeri naturali  $\mathbb{N}$ , allora  $\mathcal{R}$  è l'ordinamento  $\leq$  che conosciamo. Se  $\mathbb{D}$  fosse l'insieme di tutte le sequenze finite di caratteri dell'alfabeto, allora  $\mathcal{R}$  sarebbe l'ordine lessicografico tra sequenze di caratteri, ovvero l'ordine che usiamo nei dizionari.

### 6.8.1 Ordinamenti come visita di uno spazio di configurazioni

Può essere utile richiamare l'idea che programmare significhi individuare un procedimento che, data una qualsiasi istanza di un problema, sia in grado di muoversi da un configurazione iniziale ad una finale nell'opportuno spazio di configurazioni. Nel caso dell'ordinamento di numeri la visita deve percorrere un cammino tra configurazioni in cui si scambino via via valori, in modo da raggiungere la configurazione in cui tutti i valori della configurazione iniziale siano riorganizzati ad esempio, in ordine crescente.

Se, come esempio di riferimento, ci ponessimo il problema di ordinare le componenti di  $(4,3,2,0,1)$ , allora dovremmo trovare un cammino all'interno delle  $5!$  permutazioni di elementi seguente che vada dalla configurazione  $(4,3,2,0,1)$  in riga 13 e colonna 9 fino alla configurazione  $(0,1,2,3,4)$  in riga e colonna 1:

|                 |                 |                 |                 |                 |                 |                 |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| (0, 1, 2, 3, 4) | (0, 1, 2, 4, 3) | (0, 1, 3, 2, 4) | (0, 1, 3, 4, 2) | (0, 1, 4, 2, 3) | (0, 1, 4, 3, 2) | (0, 2, 1, 3, 4) | (0, 2, 1, 4, 3) | (0, 2, 3, 1, 4) | (0, 2, 3, 4, 1) |
| (0, 2, 4, 1, 3) | (0, 2, 4, 3, 1) | (0, 3, 1, 2, 4) | (0, 3, 1, 4, 2) | (0, 3, 2, 1, 4) | (0, 3, 2, 4, 1) | (0, 3, 4, 1, 2) | (0, 3, 4, 2, 1) | (0, 4, 1, 2, 3) | (0, 4, 1, 3, 2) |
| (0, 4, 2, 1, 3) | (0, 4, 2, 3, 1) | (0, 4, 3, 1, 2) | (0, 4, 3, 2, 1) | (1, 0, 2, 3, 4) | (1, 0, 2, 4, 3) | (1, 0, 3, 2, 4) | (1, 0, 3, 4, 2) | (1, 0, 4, 2, 3) | (1, 0, 4, 3, 2) |
| (1, 2, 0, 3, 4) | (1, 2, 0, 4, 3) | (1, 2, 3, 0, 4) | (1, 2, 3, 4, 0) | (1, 2, 4, 0, 3) | (1, 2, 4, 3, 0) | (1, 3, 0, 2, 4) | (1, 3, 0, 4, 2) | (1, 3, 2, 0, 4) | (1, 3, 2, 4, 0) |
| (1, 3, 4, 0, 2) | (1, 3, 4, 2, 0) | (1, 4, 0, 2, 3) | (1, 4, 0, 3, 2) | (1, 4, 2, 0, 3) | (1, 4, 2, 3, 0) | (1, 4, 3, 0, 2) | (1, 4, 3, 2, 0) | (2, 0, 1, 3, 4) | (2, 0, 1, 4, 3) |
| (2, 0, 3, 1, 4) | (2, 0, 3, 4, 1) | (2, 0, 4, 1, 3) | (2, 0, 4, 3, 1) | (2, 1, 0, 3, 4) | (2, 1, 0, 4, 3) | (2, 1, 3, 0, 4) | (2, 1, 3, 4, 0) | (2, 1, 4, 0, 3) | (2, 1, 4, 3, 0) |
| (2, 3, 0, 1, 4) | (2, 3, 0, 4, 1) | (2, 3, 1, 0, 4) | (2, 3, 1, 4, 0) | (2, 3, 4, 0, 1) | (2, 3, 4, 1, 0) | (2, 4, 0, 1, 3) | (2, 4, 0, 3, 1) | (2, 4, 1, 0, 3) | (2, 4, 1, 3, 0) |
| (2, 4, 3, 0, 1) | (2, 4, 3, 1, 0) | (3, 0, 1, 2, 4) | (3, 0, 1, 4, 2) | (3, 0, 2, 1, 4) | (3, 0, 2, 4, 1) | (3, 0, 4, 1, 2) | (3, 0, 4, 2, 1) | (3, 1, 0, 2, 4) | (3, 1, 0, 4, 2) |
| (3, 1, 2, 0, 4) | (3, 1, 2, 4, 0) | (3, 1, 4, 0, 2) | (3, 1, 4, 2, 0) | (3, 2, 0, 1, 4) | (3, 2, 0, 4, 1) | (3, 2, 1, 0, 4) | (3, 2, 1, 4, 0) | (3, 2, 4, 0, 1) | (3, 2, 4, 1, 0) |
| (3, 4, 0, 1, 2) | (3, 4, 0, 2, 1) | (3, 4, 1, 0, 2) | (3, 4, 1, 2, 0) | (3, 4, 2, 0, 1) | (3, 4, 2, 1, 0) | (4, 0, 1, 2, 3) | (4, 0, 1, 3, 2) | (4, 0, 2, 1, 3) | (4, 0, 2, 3, 1) |
| (4, 0, 3, 1, 2) | (4, 0, 3, 2, 1) | (4, 1, 0, 2, 3) | (4, 1, 0, 3, 2) | (4, 1, 2, 0, 3) | (4, 1, 2, 3, 0) | (4, 1, 3, 0, 2) | (4, 1, 3, 2, 0) | (4, 2, 0, 1, 3) | (4, 2, 0, 3, 1) |
| (4, 2, 1, 0, 3) | (4, 2, 1, 3, 0) | (4, 2, 3, 0, 1) | (4, 2, 3, 1, 0) | (4, 3, 0, 1, 2) | (4, 3, 0, 2, 1) | (4, 3, 1, 0, 2) | (4, 3, 1, 2, 0) | (4, 3, 2, 0, 1) | (4, 3, 2, 1, 0) |

Le strategie possono essere molteplici e richiedono estensioni dello spazio di configurazioni con ulteriori componenti. In generale, lo scopo di queste ultime è distinguere la zona della configurazione in cui le componenti sono già ordinate da quella in cui l'ordine deve essere ancora stabilito. Di seguito, a titolo riassuntivo, vengono illustrate le configurazioni visitate da tre degli algoritmi di ordinamento che studieremo, nel caso si voglia ordinare l'esempio specifico (4,3,2,0,1) di configurazione.

In particolare, le configurazioni che l'algoritmo *Bubble Sort* visiterebbe sarebbero:

|                 |                 |                 |                 |                 |                 |                 |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| (0, 1, 2, 3, 4) | (0, 1, 2, 4, 3) | (0, 1, 3, 2, 4) | (0, 1, 3, 4, 2) | (0, 1, 4, 2, 3) | (0, 1, 4, 3, 2) | (0, 2, 1, 3, 4) | (0, 2, 1, 4, 3) | (0, 2, 3, 1, 4) | (0, 2, 3, 4, 1) |
| (0, 2, 4, 1, 3) | (0, 2, 4, 3, 1) | (0, 3, 1, 2, 4) | (0, 3, 1, 4, 2) | (0, 3, 2, 1, 4) | (0, 3, 2, 4, 1) | (0, 3, 4, 1, 2) | (0, 3, 4, 2, 1) | (0, 4, 1, 2, 3) | (0, 4, 1, 3, 2) |
| (0, 4, 2, 1, 3) | (0, 4, 2, 3, 1) | (0, 4, 3, 1, 2) | (0, 4, 3, 2, 1) | (1, 0, 2, 3, 4) | (1, 0, 2, 4, 3) | (1, 0, 3, 2, 4) | (1, 0, 3, 4, 2) | (1, 0, 4, 2, 3) | (1, 0, 4, 3, 2) |
| (1, 2, 0, 3, 4) | (1, 2, 0, 4, 3) | (1, 2, 3, 0, 4) | (1, 2, 3, 4, 0) | (1, 2, 4, 0, 3) | (1, 2, 4, 3, 0) | (1, 3, 0, 2, 4) | (1, 3, 0, 4, 2) | (1, 3, 2, 0, 4) | (1, 3, 2, 4, 0) |
| (1, 3, 4, 0, 2) | (1, 3, 4, 2, 0) | (1, 4, 0, 2, 3) | (1, 4, 0, 3, 2) | (1, 4, 2, 0, 3) | (1, 4, 2, 3, 0) | (1, 4, 3, 0, 2) | (1, 4, 3, 2, 0) | (2, 0, 1, 3, 4) | (2, 0, 1, 4, 3) |
| (2, 0, 3, 1, 4) | (2, 0, 3, 4, 1) | (2, 0, 4, 1, 3) | (2, 0, 4, 3, 1) | (2, 1, 0, 3, 4) | (2, 1, 0, 4, 3) | (2, 1, 3, 0, 4) | (2, 1, 3, 4, 0) | (2, 1, 4, 0, 3) | (2, 1, 4, 3, 0) |
| (2, 3, 0, 1, 4) | (2, 3, 0, 4, 1) | (2, 3, 1, 0, 4) | (2, 3, 1, 4, 0) | (2, 3, 4, 0, 1) | (2, 3, 4, 1, 0) | (2, 4, 0, 1, 3) | (2, 4, 0, 3, 1) | (2, 4, 1, 0, 3) | (2, 4, 1, 3, 0) |
| (2, 4, 3, 0, 1) | (2, 4, 3, 1, 0) | (3, 0, 1, 2, 4) | (3, 0, 1, 4, 2) | (3, 0, 2, 1, 4) | (3, 0, 2, 4, 1) | (3, 0, 4, 1, 2) | (3, 0, 4, 2, 1) | (3, 1, 0, 2, 4) | (3, 1, 0, 4, 2) |
| (3, 1, 2, 0, 4) | (3, 1, 2, 4, 0) | (3, 1, 4, 0, 2) | (3, 1, 4, 2, 0) | (3, 2, 0, 1, 4) | (3, 2, 0, 4, 1) | (3, 2, 1, 0, 4) | (3, 2, 1, 4, 0) | (3, 2, 4, 0, 1) | (3, 2, 4, 1, 0) |
| (3, 4, 0, 1, 2) | (3, 4, 0, 2, 1) | (3, 4, 1, 0, 2) | (3, 4, 1, 2, 0) | (3, 4, 2, 0, 1) | (3, 4, 2, 1, 0) | (4, 0, 1, 2, 3) | (4, 0, 1, 3, 2) | (4, 0, 2, 1, 3) | (4, 0, 2, 3, 1) |
| (4, 0, 3, 1, 2) | (4, 0, 3, 2, 1) | (4, 1, 0, 2, 3) | (4, 1, 0, 3, 2) | (4, 1, 2, 0, 3) | (4, 1, 2, 3, 0) | (4, 1, 3, 0, 2) | (4, 1, 3, 2, 0) | (4, 2, 0, 1, 3) | (4, 2, 0, 3, 1) |
| (4, 2, 1, 0, 3) | (4, 2, 1, 3, 0) | (4, 2, 3, 0, 1) | (4, 2, 3, 1, 0) | (4, 3, 0, 1, 2) | (4, 3, 0, 2, 1) | (4, 3, 1, 0, 2) | (4, 3, 1, 2, 0) | (4, 3, 2, 0, 1) | (4, 3, 2, 1, 0) |

Invece, le configurazioni che l'algoritmo *Selection Sort* visiterebbe sarebbero:

|                 |                   |                 |                 |                 |                   |                 |                 |                   |                 |
|-----------------|-------------------|-----------------|-----------------|-----------------|-------------------|-----------------|-----------------|-------------------|-----------------|
| (0, 1, 2, 3, 4) | ← (0, 1, 2, 4, 3) | (0, 1, 3, 2, 4) | (0, 1, 3, 4, 2) | (0, 1, 4, 2, 3) | (0, 1, 4, 3, 2)   | (0, 2, 1, 3, 4) | (0, 2, 1, 4, 3) | (0, 2, 3, 1, 4)   | (0, 2, 3, 4, 1) |
| (0, 2, 4, 1, 3) | (0, 2, 4, 3, 1)   | (0, 3, 1, 2, 4) | (0, 3, 1, 4, 2) | (0, 3, 2, 1, 4) | ← (0, 3, 2, 4, 1) | (0, 3, 4, 1, 2) | (0, 3, 4, 2, 1) | (0, 4, 1, 2, 3)   | (0, 4, 1, 3, 2) |
| (0, 4, 2, 1, 3) | (0, 4, 2, 3, 1)   | (0, 4, 3, 1, 2) | (0, 4, 3, 2, 1) | (1, 0, 2, 3, 4) | (1, 0, 2, 4, 3)   | (1, 0, 3, 2, 4) | (1, 0, 3, 4, 2) | (1, 0, 4, 2, 3)   | (1, 0, 4, 3, 2) |
| (1, 2, 0, 3, 4) | (1, 2, 0, 4, 3)   | (1, 2, 3, 0, 4) | (1, 2, 3, 4, 0) | (1, 2, 4, 0, 3) | (1, 2, 4, 3, 0)   | (1, 3, 0, 2, 4) | (1, 3, 0, 4, 2) | (1, 3, 2, 0, 4)   | (1, 3, 2, 4, 0) |
| (1, 3, 4, 0, 2) | (1, 3, 4, 2, 0)   | (1, 4, 0, 2, 3) | (1, 4, 0, 3, 2) | (1, 4, 2, 0, 3) | (1, 4, 2, 3, 0)   | (1, 4, 3, 0, 2) | (1, 4, 3, 2, 0) | (2, 0, 1, 3, 4)   | (2, 0, 1, 4, 3) |
| (2, 0, 3, 1, 4) | (2, 0, 3, 4, 1)   | (2, 0, 4, 1, 3) | (2, 0, 4, 3, 1) | (2, 1, 0, 3, 4) | (2, 1, 0, 4, 3)   | (2, 1, 3, 0, 4) | (2, 1, 3, 4, 0) | (2, 1, 4, 0, 3)   | (2, 1, 4, 3, 0) |
| (2, 3, 0, 1, 4) | (2, 3, 0, 4, 1)   | (2, 3, 1, 0, 4) | (2, 3, 1, 4, 0) | (2, 3, 4, 0, 1) | (2, 3, 4, 1, 0)   | (2, 4, 0, 1, 3) | (2, 4, 0, 3, 1) | (2, 4, 1, 0, 3)   | (2, 4, 1, 3, 0) |
| (2, 4, 3, 0, 1) | (2, 4, 3, 1, 0)   | (3, 0, 1, 2, 4) | (3, 0, 1, 4, 2) | (3, 0, 2, 1, 4) | (3, 0, 2, 4, 1)   | (3, 0, 4, 1, 2) | (3, 0, 4, 2, 1) | (3, 1, 0, 2, 4)   | (3, 1, 0, 4, 2) |
| (3, 1, 2, 0, 4) | (3, 1, 2, 4, 0)   | (3, 1, 4, 0, 2) | (3, 1, 4, 2, 0) | (3, 2, 0, 1, 4) | (3, 2, 0, 4, 1)   | (3, 2, 1, 0, 4) | (3, 2, 1, 4, 0) | (3, 2, 4, 0, 1)   | (3, 2, 4, 1, 0) |
| (3, 4, 0, 1, 2) | (3, 4, 0, 2, 1)   | (3, 4, 1, 0, 2) | (3, 4, 1, 2, 0) | (3, 4, 2, 0, 1) | (3, 4, 2, 1, 0)   | (4, 0, 1, 2, 3) | (4, 0, 1, 3, 2) | (4, 0, 2, 1, 3)   | (4, 0, 2, 3, 1) |
| (4, 0, 3, 1, 2) | (4, 0, 3, 2, 1)   | (4, 1, 0, 2, 3) | (4, 1, 0, 3, 2) | (4, 1, 2, 0, 3) | (4, 1, 2, 3, 0)   | (4, 1, 3, 0, 2) | (4, 1, 3, 2, 0) | ← (4, 2, 0, 1, 3) | (4, 2, 0, 3, 1) |
| (4, 2, 1, 0, 3) | (4, 2, 1, 3, 0)   | (4, 2, 3, 0, 1) | (4, 2, 3, 1, 0) | (4, 3, 0, 1, 2) | (4, 3, 0, 2, 1)   | (4, 3, 1, 0, 2) | (4, 3, 1, 2, 0) | (4, 3, 2, 0, 1)   | (4, 3, 2, 1, 0) |

Infine, le configurazioni che l'algoritmo *Insertion Sort* visiterebbe sarebbero:

|                 |                   |                   |                   |                   |                   |                   |                   |                 |                   |
|-----------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-----------------|-------------------|
| (0, 1, 2, 3, 4) | ← (0, 1, 2, 4, 3) | (0, 1, 3, 2, 4)   | (0, 1, 3, 4, 2)   | (0, 1, 4, 2, 3)   | ← (0, 1, 4, 3, 2) | (0, 2, 1, 3, 4)   | ← (0, 2, 1, 4, 3) | (0, 2, 3, 1, 4) | ← (0, 2, 3, 4, 1) |
| (0, 2, 4, 1, 3) | (0, 2, 4, 3, 1)   | (0, 3, 1, 2, 4)   | (0, 3, 1, 4, 2)   | (0, 3, 2, 1, 4)   | (0, 3, 2, 4, 1)   | (0, 3, 4, 1, 2)   | ← (0, 3, 4, 2, 1) | (0, 4, 1, 2, 3) | (0, 4, 1, 3, 2)   |
| (0, 4, 2, 1, 3) | (0, 4, 2, 3, 1)   | (0, 4, 3, 1, 2)   | (0, 4, 3, 2, 1)   | (1, 0, 2, 3, 4)   | (1, 0, 2, 4, 3)   | ← (1, 0, 3, 2, 4) | (1, 0, 3, 4, 2)   | (1, 0, 4, 2, 3) | (1, 0, 4, 3, 2)   |
| (1, 2, 0, 3, 4) | (1, 2, 0, 4, 3)   | (1, 2, 3, 0, 4)   | (1, 2, 3, 4, 0)   | ← (1, 2, 4, 0, 3) | (1, 2, 4, 3, 0)   | (1, 3, 0, 2, 4)   | (1, 3, 0, 4, 2)   | (1, 3, 2, 0, 4) | (1, 3, 2, 4, 0)   |
| (1, 3, 4, 0, 2) | (1, 3, 4, 2, 0)   | ← (1, 4, 0, 2, 3) | ← (1, 4, 0, 3, 2) | (1, 4, 2, 0, 3)   | (1, 4, 2, 3, 0)   | (1, 4, 3, 0, 2)   | (1, 4, 3, 2, 0)   | (2, 0, 1, 3, 4) | (2, 0, 1, 4, 3)   |
| (2, 0, 3, 1, 4) | ← (2, 0, 3, 4, 1) | (2, 0, 4, 1, 3)   | (2, 0, 4, 3, 1)   | (2, 1, 0, 3, 4)   | (2, 1, 0, 4, 3)   | (2, 1, 3, 0, 4)   | (2, 1, 3, 4, 0)   | (2, 1, 4, 0, 3) | (2, 1, 4, 3, 0)   |
| (2, 3, 0, 1, 4) | ← (2, 3, 0, 4, 1) | ← (2, 3, 1, 0, 4) | ← (2, 3, 1, 4, 0) | (2, 3, 4, 0, 1)   | ← (2, 3, 4, 1, 0) | (2, 4, 0, 1, 3)   | (2, 4, 0, 3, 1)   | (2, 4, 1, 0, 3) | (2, 4, 1, 3, 0)   |
| (2, 4, 3, 0, 1) | (2, 4, 3, 1, 0)   | (3, 0, 1, 2, 4)   | (3, 0, 1, 4, 2)   | (3, 0, 2, 1, 4)   | (3, 0, 2, 4, 1)   | ← (3, 0, 4, 1, 2) | (3, 0, 4, 2, 1)   | (3, 1, 0, 2, 4) | (3, 1, 0, 4, 2)   |
| (3, 1, 2, 0, 4) | (3, 1, 2, 4, 0)   | (3, 1, 4, 0, 2)   | (3, 1, 4, 2, 0)   | (3, 2, 0, 1, 4)   | (3, 2, 0, 4, 1)   | (3, 2, 1, 0, 4)   | ← (3, 2, 1, 4, 0) | (3, 2, 4, 0, 1) | (3, 2, 4, 1, 0)   |
| (3, 4, 0, 1, 2) | (3, 4, 0, 2, 1)   | (3, 4, 1, 0, 2)   | (3, 4, 1, 2, 0)   | (3, 4, 2, 0, 1)   | ← (3, 4, 2, 1, 0) | (4, 0, 1, 2, 3)   | (4, 0, 1, 3, 2)   | (4, 0, 2, 1, 3) | (4, 0, 2, 3, 1)   |
| (4, 0, 3, 1, 2) | (4, 0, 3, 2, 1)   | (4, 1, 0, 2, 3)   | (4, 1, 0, 3, 2)   | (4, 1, 2, 0, 3)   | (4, 1, 2, 3, 0)   | ← (4, 1, 3, 0, 2) | (4, 1, 3, 2, 0)   | (4, 2, 0, 1, 3) | (4, 2, 0, 3, 1)   |
| (4, 2, 1, 0, 3) | (4, 2, 1, 3, 0)   | (4, 2, 3, 0, 1)   | (4, 2, 3, 1, 0)   | (4, 3, 0, 1, 2)   | (4, 3, 0, 2, 1)   | (4, 3, 1, 0, 2)   | ← (4, 3, 1, 2, 0) | (4, 3, 2, 0, 1) | (4, 3, 2, 1, 0)   |

### 6.8.2 Bubble sort

Abbiamo già studiato il problema di far emergere il valore massimo contenuto in un array *a* di interi, scambiando via via i valori di celle adiacenti. Se riapplichiamo lo stesso meccanismo di emersione del massimo ad *a* in penultima posizione verrà spostato il secondo valore massimo in *a*. Riapplicando una terza volta il procedimento, avremo il terzo valore più alto di *a* in terzultima posizione. Ripetendo, quindi, l'emersione del massimo un numero sufficiente di volte ad *a*, alla fine l'*array* risulterà ordinato in maniera non decrescente. Siccome, l'emersione del massimo avviene scambiando, quando necessario, i valori adiacenti di elementi in *a*, l'algoritmo che ordina *a* è individuato come *Bubble Sort*, per l'analogia che sembra esserci tra l'affiorare di bolle e l'affiorare del massimo nel modo descritto.

`BubbleSortIter.java` fornisce una possibile implementazione iterativa del *Bubble Sort*. Vale la pena osservare che:

- il metodo principale `bubbleSort (int [])` non lavora *in-place*, ovvero, modificando il parametro attuale cui esso è applicato. Al contrario, esso produce una copia del parametro attuale e lavora su di essa. Tale scelta non è obbligata ed è possibile scrivere una versione *in-place* del *Bubble Sort*;

- il metodo che fa emergere il massimo dipende da parametri che delimitano la sequenza di caselle dell'*array* sulla quale operare. Il motivo è che, man mano che l'*array* viene ordinato, accantonando i massimi verso destra, la zona in cui occorre ancora far affiorare il massimo si contrae.

`BubbleSortRec.java` fornisce una possibile implementazione ricorsiva del *Bubble Sort*. Vale la pena osservare che:

- il metodo principale `bubbleSort (int [])` lavora *in-place*. Sfruttando l'*aliasing*, modifica direttamente l'*array* passato come parametro attuale.
- è opportuno encapsulare il metodo vero e proprio di ordinamento per fissare correttamente il numero di elementi dell'*array*.

### 6.8.3 Selection sort

L'Esercizio 42 è per certi versi propedeutico all'algoritmo di ordinamento *Selection Sort* che presentiamo di seguito.

Nell'esercizio si suggeriva di far emergere il massimo in un *array*, evitando di usare il metodo tipico del *Bubble Sort*. In generale, facendo ripetutamente emergere il massimo verso destra ed in una porzione via via più piccola di un *array*, se ne ottiene una permutazione ordinata in ordine non decrescente.

Specularmente, lo stesso avviene, spostando verso sinistra il valore minimo di una porzione via via decrescente di un *array*. L'algoritmo *Selection Sort* funziona proprio iterando la “immersione verso sinistra” del minimo che si trova in una porzione sempre più piccola dell'*array* iniziale da ordinare.

`SelectionSortIter.java` e `SelectionSortRec.java` ne forniscono possibili implementazioni iterative e ricorsive.

`Correttezza e costo de SelectionSort` commentano su come si possa dimostrare la correttezza e di come si comporti l'algoritmo in termini di efficienza di ordinamento.

`SelectionSplit.java` fornisce una versione iterativa ed una ricorsiva di un metodo che si può pensare come una variante del meccanismo di riorganizzazione dell'*array* in accordo col quale il *Selection sort* funziona.

### 6.8.4 Insertion sort

L'algoritmo *Insertion Sort* è caratterizzato da un insieme di caratteristiche che lo rendono generalmente interessante, se confrontato con altri della categoria *Comparison Sorts Algorithms*, tutti caratterizzati dall'ottenere un ordinamento come conseguenza del confronto tra due elementi alla volta. L'*Insertion Sort* è stabile: non esegue scambi tra celle di un *array* da ordinare con medesimo valore, quindi non esegue scambi inutili. Se il punto di inserzione di una cella non dista mai più di  $k$  celle, allora l'*Insertion Sort* ha un costo lineare nella dimensione dell'*array* da ordinare.

`InsertionSortIter.java` e `InsertionSortRic.java` forniscono, rispettivamente, un'implementazione iterativa ed una ricorsiva dell'algoritmo in questione.

#### Schema *Insertion Split*

L'algoritmo *Insertion Sort* si comporta in accordo con uno schema che può essere descritto come segue:

- lo scopo è riorganizzare un *array* in modo che, alla fine, tutti gli elementi che soddisfino una proprietà  $P$ , data, siano, ad esempio, a sinistra di tutti quelli che non la soddisfano;
- la riorganizzazione avviene spostando passo passo ogni elemento fuori posto nella posizione immediatamente a destra dell'elemento di indice più alto che soddisfi  $P$ .

Il seguente sorgente identifica lo schema iterativo generale, che chiamiamo *Insertion Split*, in accordo col quale funziona l'*Insertion Sort*, supposto di conoscere un metodo  $P$  opportuno che restituisca valori di tipo `boolean`:

```

1 public static insertSplitIter(int[] a) {
2 if (a != null) {
3 int i = 1;
4 while (i < a.length) {
5 insert(a, i); // inserisce a[i] tra a[0] e a[i-1] a destra
6 // del primo elemento che non soddisfi P
7 i = i + 1;
8 }
9 }
10 }
11
12 private static void insert(int[] a, int i) {
13 while (i > 0 && P(a, i)) {
14 scambia(a, i, i - 1);
15 i = i - 1;
16 }

```

}

Invece, il seguente sorgente realizza ricorsivamente lo schema *Insertion Split*:

```

1 public static void insertionSplitRic(int[] a) {
2 if (a != null)
3 insertionSplit(a, 1);
4 }
5
6 private static void insertionSplit(int[] a, int i) {
7 if (i < a.length) {
8 insert(a, i); // inserisce a[i] tra a[0] e a[i-1] a destra
9 // del primo elemento che non soddisfi P
10 insertionSplit(a, i + 1);
11 }
12 }
13
14 private static void insert(int[] a, int i) {
15 if (i > 0 && P(a, i)) {
16 scambia(a, i, i - 1);
17 insert(a, i - 1);
18 }
19 }

```

Per i curiosi, [Dimostrazione di correttezza parziale originale per l'Insertion Sort](#).

La Sezione 6.3.3 de [SM14, Capitolo 6] è un possibile riferimento.

### 6.8.5 Fusione di due array ordinati

[Merge.java](#), [Correttezza del Merge.java](#),

### 6.8.6 Merge sort

[MergeSort.java](#) fornisce un algoritmo che ordina due array. Il costo si comporta come la funzione  $n(\ln_2 n)$  in cui  $n$  è la dimensione dell'array da ordinare. Il costo è giustificato rappresentando lo spazio di lavoro come albero; per completare il lavoro ad ogni livello il costo globale è lineare nella dimensione  $n$  dell'array originale da ordinare. Osservando che i livelli dell'albero sono  $\ln_2 n$  si può determinare il costo indicato.

### 6.8.7 Costi di algoritmi di ordinamento

La seguente tabella confronta i costi degli algoritmi descritti:

| Nome      | Migliore   | Medio      | Peggio     | Stabile |
|-----------|------------|------------|------------|---------|
| Insertion | $n$        | $n^2$      | $n^2$      | Sì      |
| Selection | $n^2$      | $n^2$      | $n^2$      | No      |
| Bubble    | $n$        | $n^2$      | $n^2$      | Sì      |
| Merge     | $n \log n$ | $n \log n$ | $n \log n$ | Sì      |

La sezione “*Discussion*” de [Sorting Algorithm Animations](#) riassume in maniera essenziale il motivo per cui esistono molti algoritmi di ordinamento, elencando quali siano le proprietà che si vorrebbero dall'algoritmo di ordinamento perfetto che, come viene sottolineato, non sembra esistere, proprio perché il costo dell'ordinamento dipende dall'algoritmo, dalle condizioni iniziali della sequenza da ordinare e dall'applicazione prevista per l'algoritmo.

[Algorithmic Complexity and Big-O Notation](#) offre un confronto diretto tra grafici di funzioni che tipicamente vengono usate per valutare l'efficienza di algoritmi. Ad esempio, si può notare che se il costo di un algoritmo è descrivibile con una funzione della forma  $n \log n$ , al crescere  $n$  della dimensione dell'input (ad esempio la lunghezza di un *array* da ordinare), allora esso sarà molto più efficiente di un algoritmo che risolve lo stesso problema, ma con un costo che cresce come  $n^2$ .

### 6.8.8 Eventuali approfondimenti su Algoritmi di ordinamento (per curiosi)

**Esercizio 49 (Ordinamenti non necessariamente di numeri)** Scrivere una classe con tre metodi di ordinamento `char[] bubbleSort(char[] a)`, `char[] selectionSort(char[] a)`, `char[] insertionSort(char[] a)`

a) che restituiscano una copia dell'argomento  $a$ , ordinata in accordo con l'ordine lessico grafico, definito come segue. L'alfabeto  $\Sigma = \{a, b, \dots, z\}$  è ordinato totalmente come segue  $a < b < \dots < z$ . Date due sequenze di simboli, presi in  $\Sigma$ :

$$I = \delta_{i_1} \delta_{i_2} \dots \delta_{i_n}$$

$$J = \delta_{j_1} \delta_{j_2} \dots \delta_{j_m}$$

diciamo che  $I < J$  se esiste un numero  $k \in \mathbb{N}$  per cui  $\delta_{i_1} \delta_{i_2} \dots \delta_{i_k} = \delta_{j_1} \delta_{j_2} \dots \delta_{j_k}$  e vale  $\delta_{i_{k+1}} < \delta_{j_{k+1}}$  oppure  $n = k < m$ .

Chiave di ogni metodo dovrà essere il metodo boolean minoreUguale(char[] a, char[] b) in grado di restituire true se la sequenza di caratteri in  $a$  è minore o uguale a quella in  $b$  in accordo con l'ordine lessico-grafico. Per definire boolean minoreUguale(char[] a, char[] b) è ammesso usare il metodo public static int compare(char x, char y) della classe Character. ■

## 6.9 Approfondimenti eventuali su *Array* (NON è parte del programma didattico)

### 6.9.1 Iterazione e ricorsione su *array*

Il problema di stampare tutte le permutazioni di  $n$  elementi è interessante perché può essere svolto mescolando in maniera naturale ricorsione e iterazione su un *array* di elementi.

Partiamo ragionando in maniera astratta. Supponiamo di avere a disposizione cinque elementi 6, 1, 3, 5 e 8. Lo scopo è scrivere la sequenza di sequenze:

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 61358 | 16358 | 36158 | 56138 | 86135 |
| 61385 | 16385 | 36185 | 56183 | 86153 |
| 61538 | 16538 | 36518 | 56318 | 86315 |
| 61583 | 16583 | 36581 | 56381 | 86351 |
| 61835 | 16835 | 36815 | 56813 | 86513 |
| 61853 | 16853 | 36851 | 56831 | 86531 |
| 63158 | 13658 | 31658 | 51638 | 81635 |
| 63185 | 13685 | 31685 | 51683 | 81653 |
| 63518 | 13568 | 31568 | 51368 | 81365 |
| 63581 | 13586 | 31586 | 51386 | 81356 |
| 63815 | 13865 | 31865 | 51863 | 81563 |
| 63851 | 13856 | 31856 | 51836 | 81536 |
| 65138 | 15638 | 35618 | 53618 | 83615 |
| 65183 | 15683 | 35681 | 53681 | 83651 |
| 65318 | 15368 | 35168 | 53168 | 83165 |
| 65381 | 15386 | 35186 | 53186 | 83156 |
| 65813 | 15863 | 35861 | 53861 | 83561 |
| 65831 | 15836 | 35816 | 53816 | 83516 |
| 68135 | 18635 | 38615 | 58613 | 85613 |
| 68153 | 18653 | 38651 | 58631 | 85631 |
| 68315 | 18365 | 38165 | 58163 | 85163 |
| 68351 | 18356 | 38156 | 58136 | 85136 |
| 68513 | 18563 | 38561 | 58361 | 85361 |
| 68531 | 18536 | 38516 | 58316 | 85316 |

Una strategia è la seguente. Supponiamo la sequenza iniziale sia contenuta in un *array*:

| [0] | [1] | [2] | [3] | [4] |
|-----|-----|-----|-----|-----|
| 6   | 1   | 3   | 5   | 8   |

Possiamo osservare che la prima colonna di permutazioni è stata ottenuta dall'*array* iniziale, fissando il primo elemento in posizione 0 e generando tutte le permutazioni degli elementi di posizione 1, 2, 3 e 4. Se già tentassimo una formalizzazione abbiamo appena detto d'aver generato le permutazioni di  $4 = n - 1$  elementi disponibili contenuti nella porzione d'*array*  $a[1 \dots a.length - 1]$ .

La seconda colonna di permutazioni è ottenuta in accordo con lo stesso principio, ovvero permutando tutti gli elementi in  $a[1 \dots a.length - 1]$ , ma solo dopo aver portato  $a[1]$  in prima posizione, scambiandolo con  $a[0]$ .

Per la terza colonna vale discorso analogo. Essa è ottenuta permutando tutti gli elementi in  $a[1 \dots a.length - 1]$ , ma solo dopo aver portato  $a[2]$  in prima posizione, scambiandolo con  $a[0]$ .

Il modo in cui le restanti colonne di permutazioni ripete lo schema generico seguente. La  $i$ -esima colonna è ottenuta permutando tutti gli elementi in posizione  $a[1 \dots a.length - 1]$  dopo aver portato  $a[i-1]$  in prima posizione, scambiandolo con  $a[0]$ .

Una prima approssimazione di formalizzazione è nel seguente listato:

```

1 for (int i = 0 ; i < a.length ; i++) {
2 swap(a, 0, i); // scambio di a[0] e a[i]
3 perm(a, 1);
4 // rientro dopo perm(a, 1)
5 }

```

In esso cominciamo con lo scambiare  $a[0]$  con se stesso. Ovviamente è un passo inutile, ma scriviamo l'algoritmo in questo modo per maggiore uniformità. Dopo lo scambio, per ora fittizio, generiamo le permutazioni degli elementi in  $a[1 \dots a.length - 1]$ , ottenendo la prima colonna di permutazioni. L'idea è che il metodo `perm` riorganizzi realmente gli elementi in  $a[1 \dots a.length - 1]$ . Questo significa che al rientro da `perm(a, 1)` la porzione  $a[1 \dots a.length - 1]$  sia:

| [1] | [2] | [3] | [4] |
|-----|-----|-----|-----|
| 8   | 5   | 3   | 1   |

e non:

| [1] | [2] | [3] | [4] |
|-----|-----|-----|-----|
| 1   | 3   | 5   | 8   |

come prima della chiamata a `perm(a, 1)`. Per evitare di portare  $a[i]$  in prima posizione, scambiandolo con  $a[0]$ , in una situazione inconsistente con quella iniziale, il richiamo di `perm` deve avvenire su una copia di  $a$ . Una migliore approssimazione dell'algoritmo diventa, quindi:

```

1 for (int i = 0 ; i < a.length ; i++) {
2 swap(a, 0, i); // scambio di a[0] e a[i]
3 perm(cloneArray(a), 1);
4 }

```

A questo punto basta un ulteriore passo di generalizzazione.

Cosa succede quando richiamiamo `perm(cloneArray(a), 1)`?

Ricominciamo il procedimento già visto, ma da una posizione più a sinistra in  $a$ . Ovvero, eseguiremmo:

```

2 for (int i = 1 ; i < a.length ; i++) {
3 swap(a, 1, i); // scambio di a[0] e a[i]
4 perm(cloneArray(a), 2);

```

intendendo portare  $a[i]$  in prima posizione, ovvero scambiandolo con  $a[1]$  (e non  $a[0]!$ ), per poi generare tutte le permutazioni della porzione di *array*  $a[2 \dots a.length - 1]$ .

Il codice corrispondente al passo generico diventa:

```

2 for (int i = start ; i < a.length ; i++) {
3 swap(a, start, i); // scambio di a[0] e a[i]
4 perm(cloneArray(a), start + 1);

```

con  $start$  che varia tra 0 e  $a.length - 1$ . In particolare, quando  $start$  coincide con  $a.length - 1$ , allora  $a$  contiene una nuova permutazione che, ad esempio, possiamo stampare. Questo conclude il ragionamento che porta alla codifica completa dell'algoritmo:

```

2 public static void perm(int[] a, int start) {
3 if (start == a.length - 1) {
4 log(a); // stampa a dal primo all'ultimo elemento
5 } else
6 for (int other = start ; other < a.length ; other++) {
7 swap(a, start, other); // scambia a[start] e a[other]
8 perm(cloneArray(a), start + 1); // cloneArray clona l'array a
9 }

```

La classe [PermutazioniDiInteri.java](#) completa il sorgente qui sopra definito per la stampa effettiva di permutazioni.

**Esercizio 50** Modificare l'algoritmo `perm(a, start)` per generare le disposizioni di  $k$  elementi presi tra  $n$ . ■

### 6.9.2 Strutture dati con *array*

[ArraySemipieni.java](#) è il primo esempio ragionevolmente completo di tipo di dato astratto, ovvero di una struttura che fornisce un universo di elementi su cui operare con un insieme di operazioni predefinite. Essa è costituita da una coppia di campi statici ad accesso privato. Proprio perché privati, è possibile agire sui campi solo attraverso operazioni pubbliche rese disponibili dalla classe.

Si può osservare che se, per un qualche motivo, si renda necessario progettare una classe `C.java` che usi due array parzialmente riempiti, ciascuno libero di avere la propria evoluzione, allora sarà necessario duplicare la classe di array parzialmente riempiti. Lo scopo è avere due istanze di coppie di campi privati, ciascuna con il proprio nome, per evitare sovrascritture da parte di metodi dell'altra classe. Lo schema di codice sarebbe:

```

1 public class ArraySemipieno {
2 private static int[] a0;
3 private static int p0;
4 public static void init() {
5 a0 = new int[1];
6 p0 = 0;
7 }
8
9 }
10
11 public class ArraySemipieno1 {
12 private static int[] a1;
13 private static int p1;
14 public static void init() {
15 a1 = new int[1];
16 p1 = 0;
17 }
18
19 }
20
21 public class C {
22 ArraySemipieno0.init(); // prima "istanza"
23 ArraySemipieno1.init(); // seconda "istanza"
24 }
25

```

## Pila

Opportune restrizioni e ridenomine sulla struttura degli *array* semi-pieni permettono di realizzare il tipo di dato astratto [PilaInt.java](#) che impone la gestione di valori interi in accordo con la politica L(ast)I(n)F(irst)O(ut).

**Esercizio 51** • Scrivere una classe che, sfruttando uno *stack* di caratteri `char`, abbia un metodo iterativo in grado di leggere sequenze composte dai soli caratteri '(', ')' e 'f' e sia in grado di dire se in una sequenza di parentesi, terminata con 'f', le parentesi siano correttamente aperte e chiuse.

Ad esempio, `((())((())f` è una sequenza corretta, mentre `((())(((())f` no.

([ParentesiOKPilaChar.java](#).)

- Simulare il codice della seguente classe [StrutturaDatiArrayStatico.java](#) e [StrutturaDatiArrayStaticoTest.java](#) per avere un esempio essenziale di come usare campi statici privati di per codificare tipi di dato astratto.
- Questo esercizio è un esempio di come si possano realizzare *array* parzialmente riempiti, in analogia con quanto detto in [SM14, Sezione 6.1.6].

Partendo dall'idea in [StrutturaDatiArrayStatico.java](#) definire una classe `RicercaLineareCoppia.java` con un metodo che realizzi la ricerca lineare del valore in  $k$  in un *array* di interi  $a$ . Il metodo memorizza il risultato in una coppia (di campi) tali che:

- un campo è booleano e l'altro è un intero,
- se il booleano è `true`, allora l'intero è la posizione del valore di  $k$  in  $a$ . Se il booleano è `false`, allora il valore in  $k$  non esiste in  $a$  e il campo intero deve assumere il valore  $-1$ .

(Soluzione [RicercaLineareCoppia.java](#).)

- Generalizzare `RicercaLineareCoppia.java`, scrivendo `RicercaLineareEsaustivaCoppia.java`, in modo da realizzare la ricerca lineare di ogni occorrenza del valore in `k` in un *array* di interi `a`. Il risultato deve essere strutturato in modo da contenere almeno un campo booleano ed un campo *array* di interi tale che:

- se il booleano è `true`, allora l'*array* contiene la posizione di tutte le occorrenze di `k` in `a`.
- Se il booleano è `false`, allora `k` non occorre in `a`.

(Soluzione [RicercaLineareEsaustivaCoppia.java](#).)

- Definire una classe che realizzi il tipo di dato astratto **Multi-insieme**, corredata, come da definizione, delle apposite **operazioni su multi-insiemi**.

(Proposta, certamente migliorabile, di implementazione [MultiInsiemiInt.java](#).)

- Studiare la definizione della struttura **Coda** ed implementarla per mezzo di un ***Circular buffer***

(Soluzione possibile [BufferCircolare.java](#).) ■

### 6.9.3 Indirizzamento indiretto: Crivello di Eratostene, Counting sort

Un esempio di cosa sia l'indirizzamento indiretto è fornito dalla simulazione di una possibile implementazione del `CrivelloEratostene.java` che individua numeri primi tra gli indici di un array di booleani di lunghezza fissata.

**Esercizio 52 (Riscrittura del “Crivello di Eratostene”.)** Riscrivere il metodo in: `CrivelloEratostene.java` in cui il ciclo:

```
for (int multiplo = numero * 2; multiplo <= nMax; multiplo += numero)
 primi[multiplo] = false;
```

sia sostituito con uno alternativo, ma equivalente.

(Soluzione [EratosteneAlternativo.java](#)). ■

`CountingSort.java` presenta programmi che realizzano gli omonimi algoritmi di ordinamento.

In particolare, si può osservare che il cui costo del *Counting sort* è il massimo tra il massimo valore contenuto nell'*array* `a` da ordinare e la lunghezza di `a`.



Parte II

In LABORATORIO



# Capitolo 1

## Primi passi in laboratorio

### 1.1 Il PC da linea di comando

Quando abbia senso, privilegeremo l'interazione col PC attraverso la linea di comando. Lo scopo è evidenziare l'esistenza di componenti il PC, e del sistema operativo che le gestisce, normalmente nascoste dalle interfacce grafiche.

Siccome i laboratori per il corso sono dotati di sistemi operativi Microsoft™ accenneremo agli elementi essenziali dell'interazione col PC attraverso il *command prompt* di un systema Windows™.

Ogni sistema operativo fornisce almeno un interprete per i comandi testuali:

- MS Windows ha il “Prompt dei comandi”:

```
Welcome to FreeDOS

CuteMouse v1.9.1 alpha 1 [FreeDOS]
Installed at PS/2 port
C:\>ver

FreeCom version 0.82 p1 3 XMS_Swap [Dec 18 2003 06:49:21]

C:\>dir
Volume in drive C is FREEDOS_C95
Volume Serial Number is 0E4F-19EB
Directory of C:\

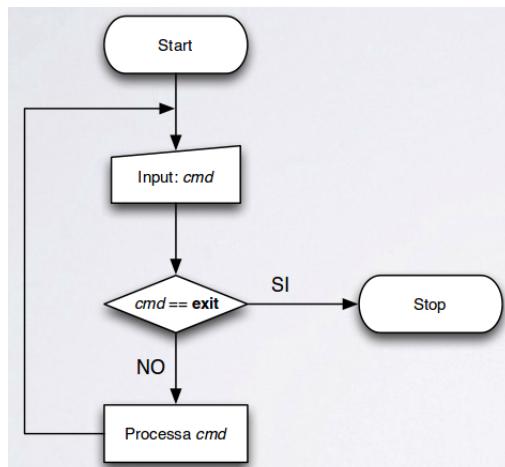
FDOS <DIR> 08-26-04 6:23p
AUTOEXEC.BAT 435 08-26-04 6:24p
BOOTSECT.BIN 512 08-26-04 6:23p
COMMAND.COM 93,963 08-26-04 6:24p
CONFIG.SYS 801 08-26-04 6:24p
FDOSBOOT.BIN 512 08-26-04 6:24p
KERNEL.SYS 45,815 04-17-04 9:19p
 6 file(s) 142,038 bytes
 1 dir(s) 1,064,517,632 bytes free

C:\>_
```

- Sistemi operativi à la UNIX, come ad esempio Linux e OSX, forniscono almeno una *shell* scelta tra varie possibili declinazioni: *bash*, *sh*, *csh* , ....

#### La *Command Line Interface* (CLI)

Il modo di funzionare di un interprete dei comandi, o *Command Line Interface* (CLI) funziona come illustrato nel seguente *flow-chart*:



In esso:

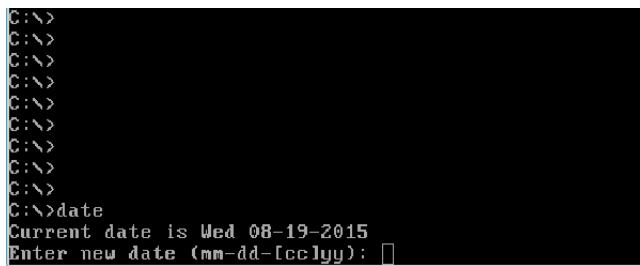
1. il programma rimane in attesa dell'immissione di un comando. L'immissione corrisponde alla scrittura di un qualche specifico comando, tra quelli effettivamente interpretabili, terminata dalla pressione del tasto "Invio", "Enter", "Return" o simili.
2. Se il comando impartito è `exit` o, in genere `CTRL-D`, allora l'interprete smette di funzionare e, in gergo, "restituisce il *prompt*".
3. Altrimenti, a comando impartito, l'interprete lo interpreta e, se previsto, produce un risultato visibile nelle righe seguenti il comando.

### Sintassi di comandi comuni

Nel caso non si sappia richiamare l'interprete dei comandi di testo, usando il proprio sistema operativo Microsoft<sup>TM</sup>, usare un motore di ricerca con parole chiave del tipo "*windows opening the program prompt*".

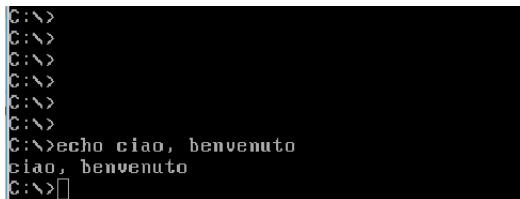
Ogni comando ha una sintassi da rispettare, pena la segnalazione di qualche errore. Esempi di semplici comandi sono:

- `date` che serve per conoscere la data in accordo con la quale il PC sta lavorando, o per modificarla:



```
C:\>
C:\>
C:\>
C:\>
C:\>
C:\>
C:\>
C:\>
C:\>date
Current date is Wed 08-19-2015
Enter new date (mm-dd-[cc]yy):
```

- `echo` che stampa messaggi sul terminale. Ad esempio, esso accetta una sequenza di caratteri come argomento:



```
C:\>
C:\>
C:\>
C:\>
C:\>
C:\>
C:\>
C:\>echo ciao, benvenuto
ciao, benvenuto
C:\>
```

Nel caso illustrato, l'argomento è composto dalla sequenza di caratteri "ciao, benvenuto".

I comandi testuali disponibili o sono *built-in*, ovvero fanno parte integrante degli strumenti resi disponibili dal sistema operativo che gestisce le risorse del PC, oppure sono messi a disposizione solo previa loro installazione. Nel secondo caso, essi sono *file*, memorizzati in particolari cartelle, dette anche *folder*, o *direcotry*, che l'interprete carica in memoria ed esegue.

#### 1.1.1 Struttura del *File system* in forma testuale

Il *file system* è una struttura logica che un PC mette a disposizione con lo scopo di archiviare permanentemente, in accordo con una struttura gerarchica, i dati sotto forma di sequenze opportune di informazioni, dette *file*.

Il seguente *screenshot* affianca due possibili visualizzazioni del *folder* `C:\windows` su di un sistema Microsoft<sup>TM</sup>:

```
C:\windows>dir
Volume in drive C has no label.
Volume Serial Number is 0000-0000

Directory of C:\windows

5/16/2014 1:33 PM <DIR> .
5/16/2014 1:33 PM <DIR> ..
5/16/2014 1:33 PM <DIR> command
8/19/2015 11:27 AM 5,540 explorer.exe
5/16/2014 1:33 PM <DIR> Fonts
5/16/2014 1:33 PM <DIR> help
8/19/2015 11:27 AM 1,032 hh.exe
5/16/2014 1:33 PM <DIR> inf
5/16/2014 1:33 PM <DIR> Installer
5/16/2014 1:33 PM <DIR> logs
5/16/2014 1:33 PM <DIR> Microsoft.NET
5/16/2014 1:33 PM <DIR> mono
8/19/2015 11:27 AM 178,336 notepad.exe
8/19/2015 11:27 AM 360,924 regedit.exe
8/19/2015 11:27 AM 212 rundll.exe
5/16/2014 1:33 PM <DIR> system
5/16/2014 1:33 PM 481 system.ini
8/19/2015 11:27 AM <DIR> system32
5/16/2014 1:33 PM <DIR> temp
8/19/2015 11:27 AM 1,032 twain_32.dll
8/19/2015 11:27 AM 206 twain.dll
8/19/2015 11:27 AM 2,218 win.ini
8/19/2015 11:27 AM 214 winhelp.exe
8/19/2015 11:27 AM 131,772 winhlp32.exe
5/16/2014 1:33 PM <DIR> winsxs
 11 files 681,967 bytes
 14 directories 34,830,344,192 bytes free

C:\windows>
```

In entrambe le visuali, è possibile individuare la sequenza di caratteri C:. Essa costituisce un nome logico per un disco rigido che memorizza in maniera permanente i dati. Altre etichette tipiche sono D:, F: e Z:.

In entrambe le visuali, l'etichetta C: compare, in realtà, nella sequenza: C:\windows. La sequenza di caratteri \windows indica il nome di una delle cartelle memorizzate sul disco C:. Lo si evince dalla finestra più a destra, nella quale possiamo leggere, in un sol colpo, molte informazioni:

- il disco C:, oltre a \windows, contiene anche altri due *folder* \Program Files e \users. Si dice che \windows, \Program Files e \users sono allo stesso livello nella gerarchia del *file system*.
- La cartella \windows contiene *files* come explorer.exe e *folder* come \system32.
- Ad un livello inferiore, rispetto a \windows, \Program Files e \users si trovano tutti i *folder* ed i *file* che esse possano contenere:
  - \drivers e \mui sono i primi due *folder* elencati come contenuto della cartella \system32,
  - \Common Files e \Internet Explorer sono due *folder* in \Program Files.

Nella finestra di sinistra, la visione del *file system* è più piatta. Il comando dir, immesso e visibile nella prima linea, ha prodotto l'elenco di *folder* e *file* presenti nella *working directory*, o *working folder*.

### Working folder

Esso è il *folder* cui hanno accesso diretto i programmi che potremmo voler eseguire tramite linea di comando. Possiamo leggere il *working folder* nella sequenza di caratteri che seguono il nome, ad esempio C:, del disco. Nella finestra sinistra descritta la sequenza C:\windows indica proprio che il *working folder* sia \windows. Il comando chdir .., cambia il *working folder* nella radice \ del *file system* sul disco C:. Lo apprezziamo per mezzo dalla seguente finestra:

```

5/16/2014 1:33 PM <DIR> logs
5/16/2014 1:33 PM <DIR> Microsoft.NET
5/16/2014 1:33 PM <DIR> mono
8/19/2015 11:27 AM 178,336 notepad.exe
8/19/2015 11:27 AM 360,924 regedit.exe
8/19/2015 11:27 AM 212 rundll.exe
5/16/2014 1:33 PM <DIR> system
5/16/2014 1:33 PM 481 system.ini
8/19/2015 11:27 AM <DIR> system32
5/16/2014 1:33 PM <DIR> temp
8/19/2015 11:27 AM 1,032 twain_32.dll
8/19/2015 11:27 AM 206 twain.dll
8/19/2015 11:27 AM 2,218 win.ini
8/19/2015 11:27 AM 214 winhelp.exe
8/19/2015 11:27 AM 131,772 winhlp32.exe
5/16/2014 1:33 PM <DIR> winsxs
 11 files 681,967 bytes
 14 directories 34,830,090,240 bytes free

C:\windows>cd ..

C:>dir
Volume in drive C has no label.
Volume Serial Number is 0000-0000

Directory of C:\

5/16/2014 1:33 PM <DIR> Program Files
5/16/2014 1:33 PM <DIR> users
5/16/2014 1:33 PM <DIR> windows
 0 files 0 bytes
 3 directories 34,830,090,240 bytes free

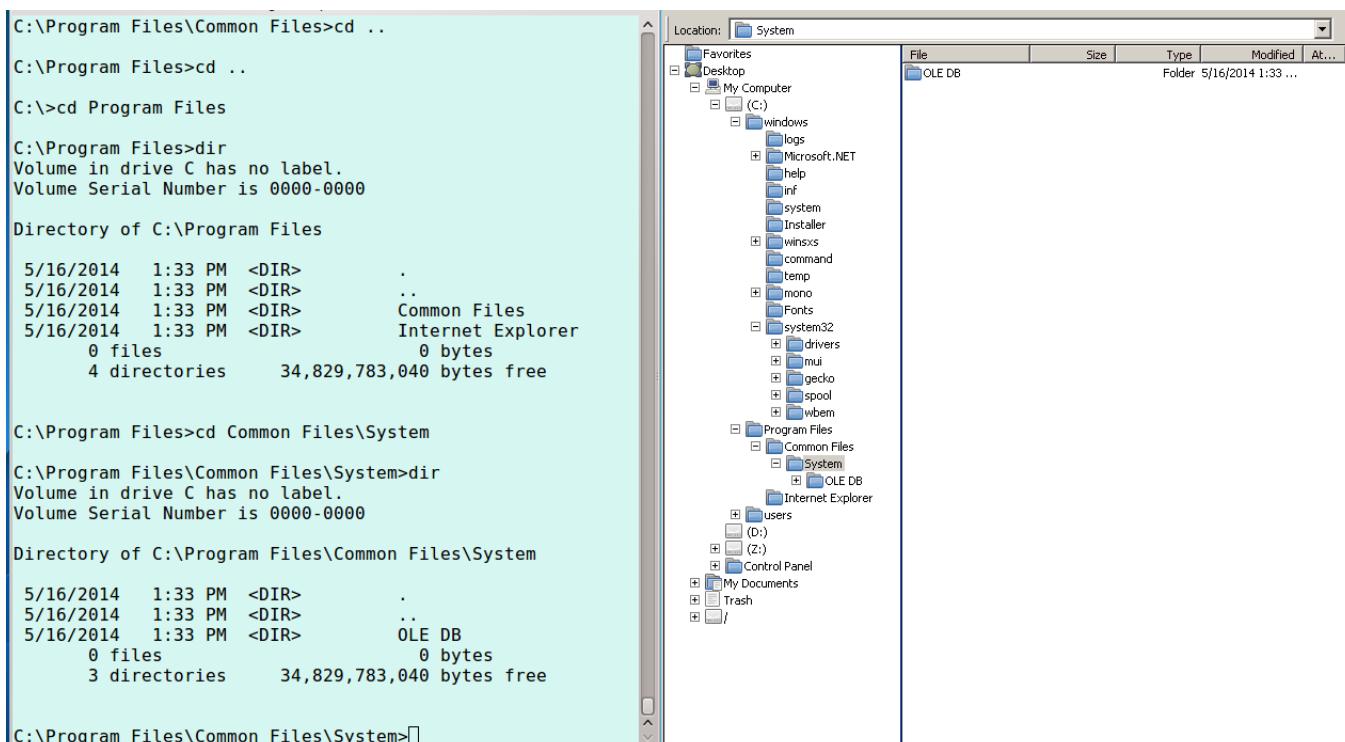
C:>■

```

Dopo cd .., il comando dir ha infatti elencato i tre *folder* Program Files, users e windows.

Più in generale, cd .. cambia la *working directory* da quella attuale al *folder* che la contiene, ovvero ci si muove verso la radice del *file system* o, equivalentemente, al padre del *folder* da cui partiamo. Il comando cd è equivalente a cd. Quindi, nel nostro caso, chdir .. cambia la *working directory* da quella attuale al *folder*. Useremo cd e chdir in modo intercambiabile.

Per “scendere” nella gerarchia delle directory il comando è sempre cd, ma ad esso va fornito il *folder* cui occorre scendere come argomento. La seguente finestra a sinistra fornisce un esempio di come procedere:



```

C:\Program Files\Common Files>cd ..
C:\Program Files>cd ..
C:\>cd Program Files
C:\Program Files>dir
Volume in drive C has no label.
Volume Serial Number is 0000-0000

Directory of C:\Program Files

5/16/2014 1:33 PM <DIR> .
5/16/2014 1:33 PM <DIR> ..
5/16/2014 1:33 PM <DIR> Common Files
5/16/2014 1:33 PM <DIR> Internet Explorer
 0 files 0 bytes
 4 directories 34,829,783,040 bytes free

C:\Program Files>cd Common Files\System
C:\Program Files\Common Files\System>dir
Volume in drive C has no label.
Volume Serial Number is 0000-0000

Directory of C:\Program Files\Common Files\System

5/16/2014 1:33 PM <DIR> .
5/16/2014 1:33 PM <DIR> ..
5/16/2014 1:33 PM <DIR> OLE DB
 0 files 0 bytes
 3 directories 34,829,783,040 bytes free

C:\Program Files\Common Files\System>■

```

Dopo due occorrenze del comando cd .. che ci hanno portato in \, ci siamo mossi in \Program Files. Da \Program Files, in un sol colpo, siamo scesi di due livelli nella gerarchia del *file system* col comando cd Common Files\System. La *working directory* contiene il *folder* OLE DB. La finestra a destra offre in un sol colpo la struttura che, con la linea di comando, vediamo solo “livello per livello”.

In generale `cd folder1\folder2\...\folderN` è il comando per muoversi dalla *working directory* ad un folder di nome `folderN` che si trovi annidata un certo numero di livelli al di sotto di `folder1`, di `folder2`, etc. . La sequenza `folder1\folder2\...\folderN` è detta *pathname* e semplicemente *path*.

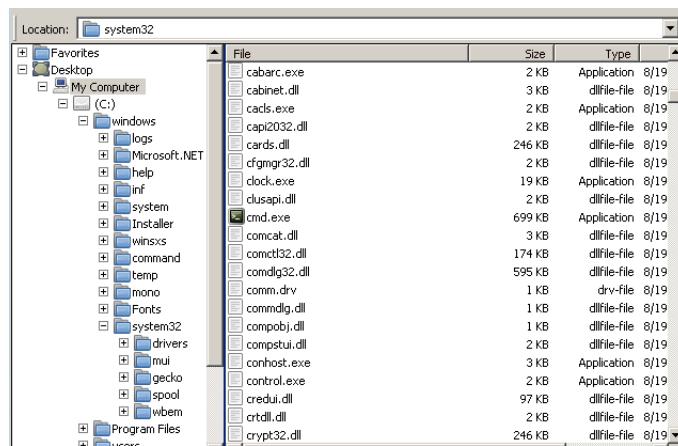
### Comandi testuali tipici per operare sul *File system*

Comandi fondamentali per la gestione del file system sono:

- `rm <nome file>` rimuove il *file* con nome `<nome file>`.
- `rmdir <nome folder>` rimuove il *folder* con nome `<nome folder>`.
- `cp <nome file sorgente> <nome file oggetto>` genera un nuovo *file* di nome `<nome file oggetto>` con contenuto identico a quello di nome `<nome file sorgente>`. Il *file* `<nome file sorgente>` rimane nel *file system* così com'è.
- `mv <nome file sorgente> <nome file oggetto>` genera un nuovo *file* di nome `<nome file oggetto>` con contenuto identico a quello di nome `<nome file sorgente>`. Il *file* `<nome file sorgente>` viene rimosso. Il risultato finale consiste nel muovere il *file* `<nome file sorgente>` nel file `<nome file oggetto>`.

#### 1.1.2 Strumenti per la codifica

Nel *file system* i *file* sono individuati sia per mezzo del *nome*, sia dell'*estensione* ad essi assegnati. La seguente figura:



elenca alcuni dei file che possono essere presenti nella cartella `C:\windows\system32` di un sistema operativo Microsoft<sup>TM</sup>. Notiamo, ad esempio, i *file* `acledit.dll`, `cmd.exe` e `comm.drv`. Il nome “proprio” del file consiste nella sequenza di caratteri che precede il punto. L'estensione del file consiste nella sequenza che segue il punto.

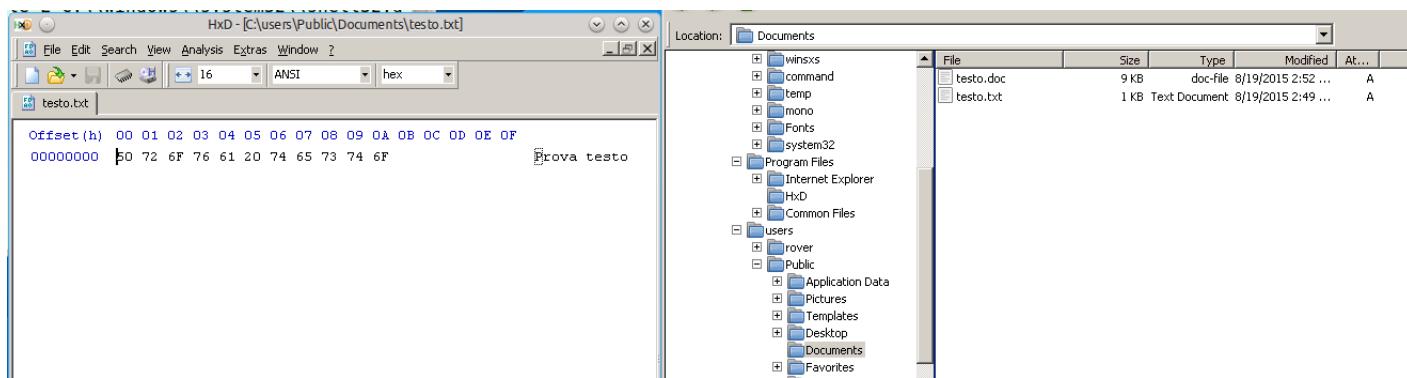
L'estensione serve ad identificare il formato in accordo col quale il *file* memorizza dati. Conseguentemente, l'estensione è utile al sistema operativo stesso per individuare l'insieme di programmi adatti a leggere, ad esempio, i dati nel *file* stesso. In sistemi operativi Microsoft<sup>TM</sup>, l'estensione `.exe` indica che il file è di tipo “binario” e contiene un programma adatto ad essere interpretato dal sistema operativo stesso, come se fosse un comando. Per comprendere il significato delle altre due estensioni `.dll` e `.drv` occorrono nozioni che è prematuro introdurre.

#### Le estensioni `txt` e `doc`

Esse sono due estensioni associate a *file* destinati a contenere documenti con testi leggibili da esseri umani. La differenza, però, è sostanziale.

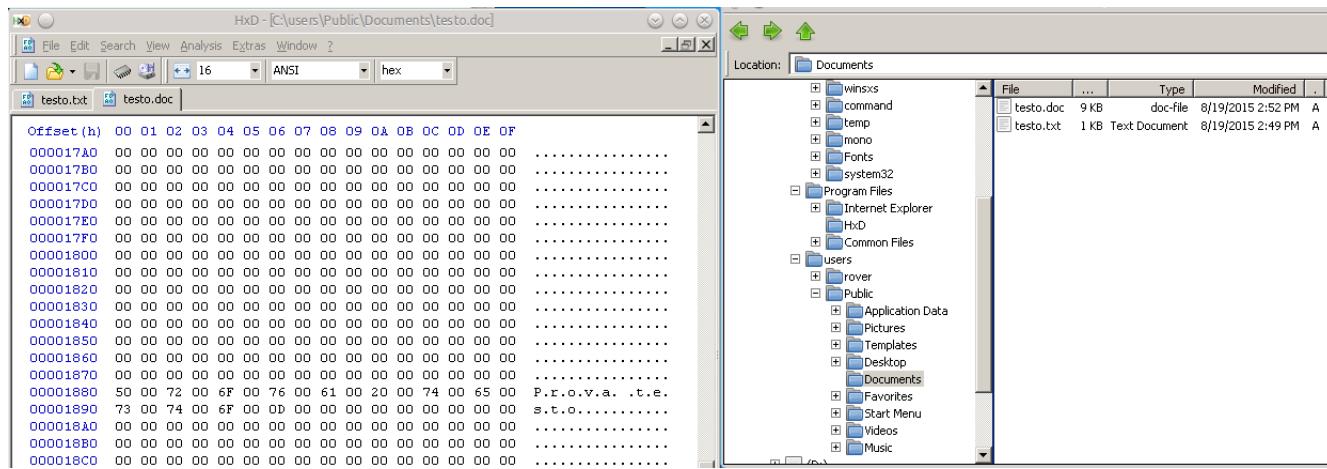
L'estensione `.txt` indica *file* che contengono, una dopo l'altra, solo le rappresentazioni di ciascuno dei caratteri che compongano il testo.

Qui sotto, a sinistra abbiamo la “fotografia” dell’interfaccia grafica dell’*editor* di testi `HxD` mentre permette la visualizzazione del *file* `testo.txt`, presente nel *file system*, come da figura a destra:



I caratteri neri nella finestra a sinistra sono gli unici che ci interessano. La sequenza esadecimale 50 72 6F 76 61 20 74 65 73 74 6F è la rappresentazione essenzialmente fedele di come la sequenza di caratteri Prova testo sia effettivamente memorizzata nel *file system* all'interno della cartella C:\users\Documents.

Confrontiamo la precedente coppia di figure con la seguente:



A sinistra, l'interfaccia dell'*editor* di testi HxD mostra una piccola porzione del contenuto di testo.doc anch'esso memorizzato nel *file system*. La differenza col precedente è sostanziale. In testo.doc troviamo molti più caratteri dello stretto necessario. Il motivo sta nella funzione che file con estensione doc debbano assumere. Essi possono contenere testi organizzati in maniera sofisticata, con testo in grassetto, strutturato per punti numerati con icone, etc. Una tale strutturazione è mantenuta codificandola opportunamente. I caratteri “in più” hanno esattamente lo scopo di rappresentare le informazioni sulla strutturazione del testo.

*Per programmare, le informazioni in più sulla struttura del testo, analoghe a quelle che si possano trovare nei file con estensione doc, non sono ammesse!*

*Per programmare, servono solo editor di testi in grado di produrre sequenze pure di caratteri, come quelle che abbiamo visto esistere nei file con estensione txt.*

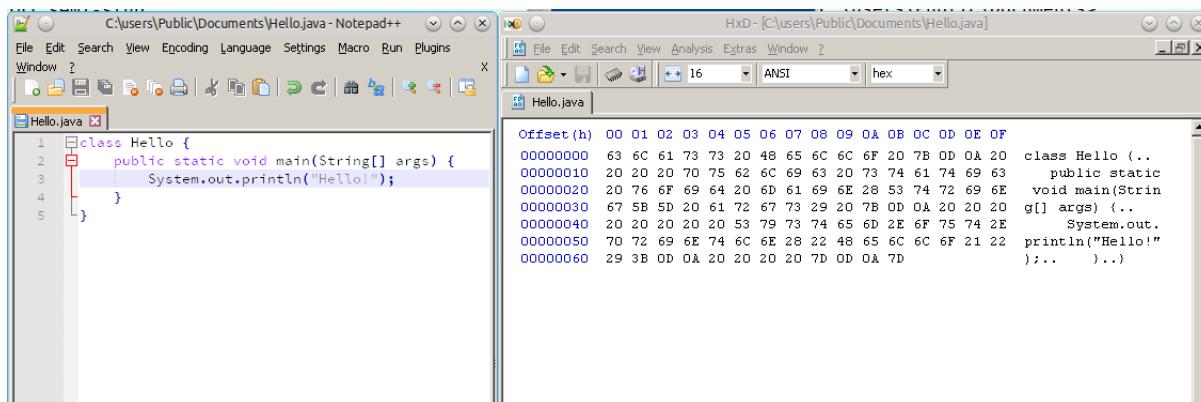
### Per i più curiosi

L'editor di testi HxD è un programma molto versatile che, come abbiamo visto, permette l'esplorazione del contenuto di file ad un livello molto basso, ovvero prossimo a quello che un PC “vede” effettivamente. Permette anche altre operazioni molto sofisticate e potenzialmente pericolose per l'integrità del *file system*. È disponibile sia nella versione di libera installazione, sia nella versione utilizzabile senza installazione, ovvero [HxD Portable](#).

### 1.1.3 NotePad++

Durante le esercitazioni in laboratorio e durante gli appelli d'esame useremo l'*editor* Notepad++ distribuito secondo la licenza [GPL](#). Esso è un *editor* che produce file di testo puri, senza codificare alcun'altra informazione se non quella necessaria a rappresentare ciascun singolo carattere di testo. Notepad++ è adatto per la programmazione con i più disparati linguaggi di programmazione perché offre strumenti ormai ritenuti necessari per la codifica di algoritmi in programmi. Una di queste è detta *syntax highlighting*; un'altra è la chiusura automatica di parentesi ed il loro posizionamento secondo convenzioni ormai standard.

Qui sotto, la figura a sinistra è un'istantanea di Notepad++ con un programma Java essenziale:



In essa si nota la colorazione delle parole, ovvero la *syntax highlighting*, che aiuta ad identificarne la funzione. A destra, HxD conferma che, nonostante il testo sia abbellito dai colori, nel file Hello.java esistano solo i caratteri necessari e nulla di più.

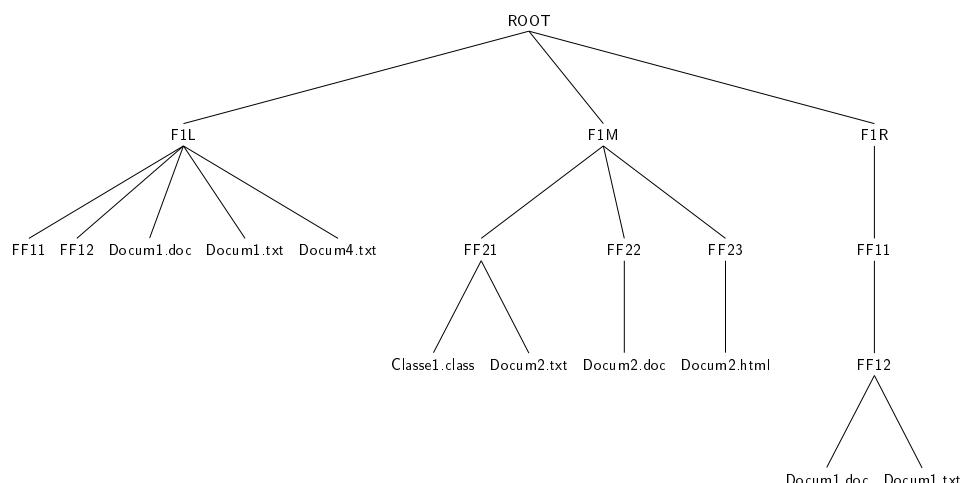
Un’ultima osservazione è per l’estensione `.java`. Essa è riservata per i programmi che rispettino la sintassi imposta dal linguaggio di programmazione omonimo e la convenzione è che essi siano *file* di testo essenziali. Il motivo per usare l’estensione `.java` è proprio facilitare la loro individuazione da parte del sistema operativo affinché, ad esempio, un “doppio click” ne permetta l’apertura automatica con un *editor* come [Notepad++](#).

**Configurazione obbligatoria al primo utilizzo.** Al primo utilizzo come nuovo utente è obbligatorio impostare il funzionamento di [Notepad++](#) come segue, dopo la sua “apertura”. Supponendo l’installazione della versione in lingua inglese: Menù principale -> “*Settings*” -> “*Preferences*” -> “*Tab Settings*” -> check su “*Replace by space*”.

Il motivo per cui impostare la configurazione descritta è che essa preserva la leggibilità del documento quando questo viene aperto con *editor* di testo diversi da [Notepad++](#)

Spaziature nel codice Java che risulteranno errate nell’*editor* del docente perché lo studente non ha seguito la procedura descritta saranno penalizzate durante la valutazione degli esercizi.

**Esercizio 53** • Scegliere un *folder* di lavoro ed impartire gli opportuni comandi testuali per generare la struttura seguente:



Quando è noto un programma che possa generare *file* con una specifica estensione, usarlo. I *file* con estensioni per le quali non sia noto il programma adatto, usare l’*editor* [Notepad++](#) e salvare specificando l’opportuna estensione.

- Riportarsi nel *folder* ROOT. Prevedere e poi sperimentare quale sarà l’effetto dei seguenti comandi sulla struttura ad albero appena costruita o sui *file* in esso contenuti:

- comp F1L\Docum1.txt F1L\Docum4.txt, magari dopo aver scritto qualche carattere in Docum1.txt e Docum4.txt.
- deltree F1R\FF11,
- cd F1M seguito da edit FF23\Docum2.html,
- edit ..\F1L\Docum4.txt e scrivere la sequenza ciao. Ripetere la stessa operazione, ma con edit ..\F1L\Docum1.txt. Quindi, provare il comando find ciao ..\F1L\Docum\*.txt e provare a dedurre il significato del simbolo \* contenuto nel secondo argomento di find.

- Dare un'occhiata alla seguente [Lista di comandi di MS-DOS](#) e sperimentare l'effetto di quelli che possano incuriosire di più, come, ad esempio `mem`, `more`, `path`, `ren`, `tree`, etc. .

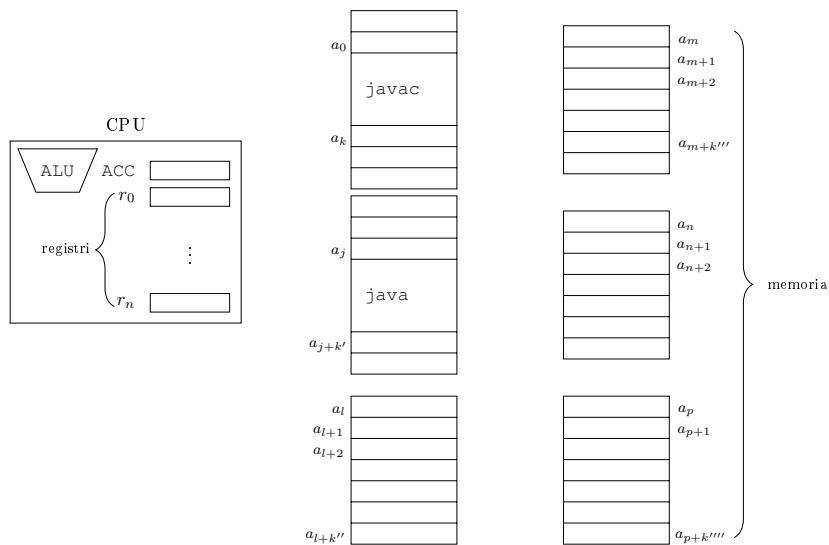
Se, per qualche comando, la breve spiegazione fornita non è sufficiente a capire quel che il comando faccia, provare a cercare on-line la sintassi d'uso ed illustrazioni migliori.

## 1.2 Compilazione e interpretazione (a.a. 17/18 leggere)

Produrre un programma, scritto nel linguaggio di riferimento Java, che il calcolatore possa correttamente utilizzare, richiede la comprensione di alcuni semplici meccanismi alla base dell'organizzazione delle informazioni da parte del PC.

### 1.2.1 Struttura essenziale di un PC.

Punto di partenza ideale è la seguente rappresentazione minimale dello stato in cui deve trovarsi il PC che intendiamo utilizzare:



**CPU.** Al sinistra campeggia una rappresentazione della *Central Processing Unit* (CPU) che interpreta programmi. In generale, essa contiene una memoria composta da *registri*  $ACC$ ,  $r_1, \dots, r_n$ . Vedremo che l'*accumulatore*  $ACC$  gioca il ruolo di registro principale. La *Arithmetic Logic Unit* (ALU) è la componente preposta all'esecuzione di semplici operazioni aritmetiche e logiche sulla rappresentazione a bassissimo livello dei dati.

**Memoria.** A lato della CPU vediamo una rappresentazione logica di varie porzioni della memoria. La rappresentazione non distingue, ad esempio, tra memoria di massa, come un *hard-disk*, e memoria centrale volatile, ad accesso diretto RAM. L'importante è immaginare che la memoria sia divisa in unità — i rettangoli — dette *byte*, ciascuna individuabile per mezzo di un unico indirizzo. In ordine crescente gli indirizzi sono  $a_0, a_1, \dots, a_k, \dots, a_j, \dots, a_{j+k'}, \dots, a_l, a_{l+1}, a_{l+2}, \dots, a_{l+k''}, \dots, a_m, a_{m+1}, \dots$

**I programmi `javac` e `java`.** Sul PC in uso, devono essere installata l'edizione standard della piattaforma Java **JDK**, disponibile su [Java SE Download](#). La piattaforma **JDK** rende disponibili i programmi `javac` e `java`, gli strumenti necessari per cominciare lo sviluppo di programmi nel linguaggio Java.

Istruzioni su come procedere per l'installazione sono disponibili sul web e sul libro di testo di riferimento, oltre che sul [JDK 8 and JRE 8 Installation Start Here](#).

### 1.2.2 Compilare un sorgente Java ed interpretare il suo “oggetto”

Supponiamo che il nostro obiettivo finale si eseguire sul nostro PC il seguente programma

```

1 public class Espressione {
2 public static void main(String[] args) {
3 int a = 3;
4 int b = 2;
5
6 int r;
7
8 r = a * a + b * a;
9 System.out.println(r);
10 }
11 }

```

memorizzato nel *file* di nome `Espressione.java`, scritto con [Notepad++](#) e che, per definizione, chiamiamo *file sorgente*.

Rispettando la sintassi del linguaggio di programmazione Java, `Espressione.java` codifica il seguente semplice algoritmo:

```

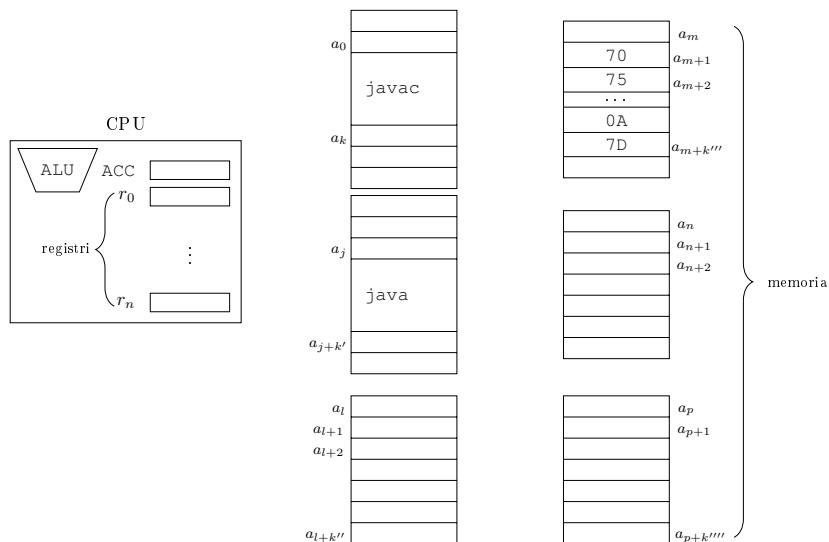
1: // a contiene 3 ∈ N
2: // b contiene 2 ∈ N
3: r = a * a + b * a;
4: stampa r;

```

il cui scopo sia d'assegnare ad `r` il risultato di una espressione, che dipende dai valori nelle variabili `a`, `b`, `c`, `d`, e di rendere noto il contenuto di `r`.

`Espressione.java` contiene diverse strutture sintattiche, che per ora, possiamo ignorare.

Il primo fatto che conti, al fine di questa sezione, è che la rappresentazione di ciascuna delle lettere che costituiscono il testo sorgente `Espressione.java` sia in qualche punto della memoria, ad esempio a partire dall'indirizzo  $a_{m+1}$ , come nello schema seguente:



La sequenza di caratteri che immaginiamo essere presente tra i *byte* di indirizzo  $a_{m+1}$  e  $a_{m+k'''}$  è:

| Offset (h) | 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F                    |
|------------|--------------------------------------------------------------------|
| 00000000   | f0 75 62 6C 69 63 20 63 6C 61 73 73 20 45 73 70 f public class Esp |
| 00000010   | 72 65 73 73 69 6F 6E 65 20 7B 0A 20 20 20 70 rression . . p        |
| 00000020   | 75 62 6C 69 63 20 73 74 61 74 69 63 20 76 6F 69 blic static voi    |
| 00000030   | 64 20 6D 61 69 6E 28 53 74 72 69 6E 67 5B 5D 20 d main(String[]    |
| 00000040   | 61 72 67 73 29 20 7B 0A 20 20 20 20 20 20 20 20 args) { .          |
| 00000050   | 69 6E 74 20 61 20 3D 20 33 3B 0A 20 20 20 20 20 20 int a = 3; .    |
| 00000060   | 20 20 20 69 6E 74 20 62 20 3D 20 32 3B 0A 20 20 int b = 2; .       |
| 00000070   | 20 20 20 20 20 20 0A 20 20 20 20 20 20 20 69 . . i                 |
| 00000080   | 6E 74 20 72 3B 0A 20 20 20 20 20 20 20 0A 20 nt r; . .             |
| 00000090   | 20 20 20 20 20 20 20 72 20 3D 20 61 20 24 20 61 r = a * a; .       |
| 000000A0   | 20 2B 20 62 20 2A 20 61 3B 0A 20 20 20 20 20 20 + b * a; .         |
| 000000B0   | 20 20 53 79 73 74 65 6D 2E 6F 75 74 2E 70 72 69 System.out.pri     |
| 000000C0   | ntln(r); . . ) .)                                                  |

ovvero, quella fornita, leggendo il sorgente con l'*editor* a basso livello [HxD](#).

Per quanto a basso livello, quindi molto vicino al linguaggio su cui una CPU opera, il sorgente `Espressione.java` tra  $a_{m+1}$  e  $a_{m+k'''}$  non è ancora nella forma adatta ad esser interpretata dal calcolatore.

Ogni sorgente (scritto in linguaggio Java) deve essere trasformato nel *file oggetto* corrispondente. La trasformazione "*sorgente* → *oggetto*" viene effettuata dal programma `javac` e passa sotto il nome di *compilazione*.

Nel nostro caso, per compilare `Espressione.java` ed ottenere il corrispondente *file oggetto*, il comando da impartire è `javac Espressione.java`, come testimoniato dalla seguente immagine:

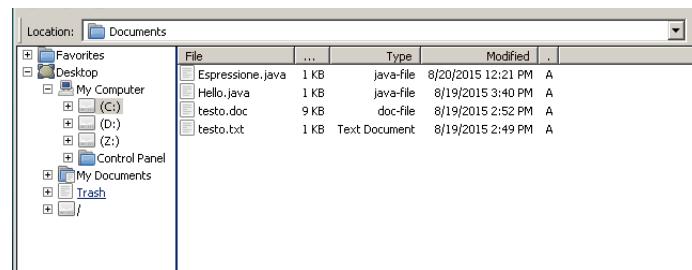
```
C:\users\Public\Documents>dir
Volume in drive C has no label.
Volume Serial Number is 0000-0000

Directory of C:\users\Public\Documents

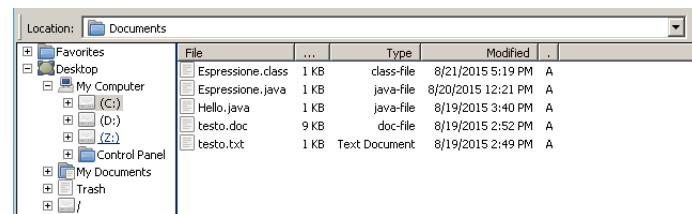
8/21/2015 4:53 PM <DIR> .
5/16/2014 1:33 PM <DIR> ..
8/20/2015 12:21 PM 246 Espressione.java
8/19/2015 3:40 PM 108 Hello.java
8/19/2015 2:52 PM 9,216 testo.doc
8/19/2015 2:49 PM 11 testo.txt
 4 files 9,581 bytes
 2 directories 34,828,627,968 bytes free

C:\users\Public\Documents>javac Espressione.java
```

Se, prima della compilazione, il *file system* contiene, i seguenti file:



dopo la compilazione esso conterrà anche il *file* oggetto Espressione.class:

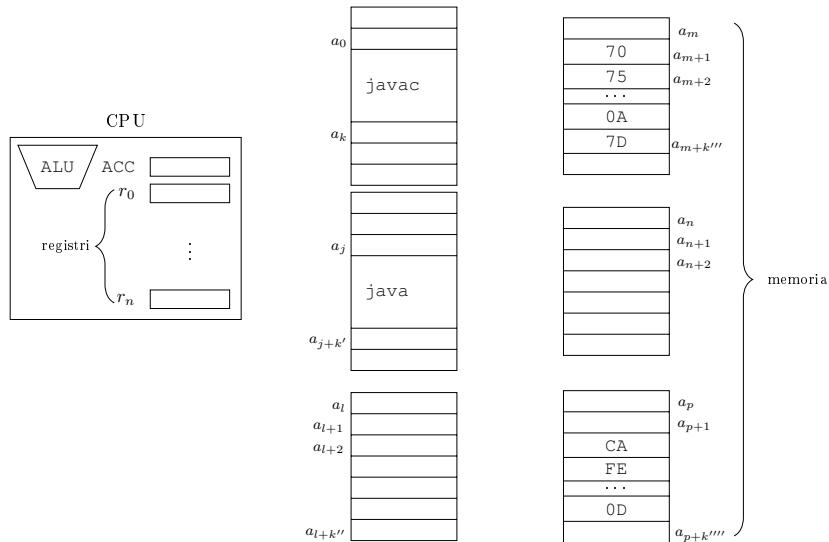


L'oggetto Espressione.class è il *file* che contiene la versione di Espressione.java che il calcolatore può interpretare. Di esso si può ottenere la seguente versione testuale, detta *disassemblata*:

```
Compiled from "Espressione.java"
public class Espressione {
 public Espressione();
 Code:
 0: aload_0
 1: invokespecial #1 // Method java/lang/Object."<init>":()V
 4: return

 public static void main(java.lang.String[]);
 Code:
 0: iconst_3
 1: istore_1
 2: iconst_2
 3: istore_2
 4: iload_1
 5: iload_1
 6: imul
 7: iload_2
 8: iload_1
 9: imul
 10: iadd
 11: istore_3
 12: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 15: iload_3
 16: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
 19: return
}
```

per mezzo del comando javap -c, fornito con la piattaforma [JDK](#). Quel che a noi veramente interessa, è che Espressione.class sia presente, rappresentato a basso livello, in qualche zona della memoria, ad esempio tra  $a_{p+2}$  e  $a_{p+k'''-1}$ :



Per definizione, il comando:

```
java Espressione
```

indica al calcolatore che il file oggetto `Espressione.class` deve essere *interpretato*.

L'*interprete* è il programma `java` che, per ipotesi, risiede in memoria tra gli indirizzi  $a_{j+1}$  e  $a_{j+k'-1}$ .

Nel nostro caso, l'effetto dell'interpretazione è “tangibile” perché, una volta interpretato, l'oggetto `Espressione.class` stampa il numero 18.

Sia la compilazione, attraverso il comando `javac Espressione.java`, sia l'interpretazione, attraverso `java Espressione`, coinvolgono la CPU:

- compilando, la CPU interpreta quel che c'è scritto nel *file* `javac` che usa il file `Espressione.java` come dato da elaborare.
- interpretando, la CPU interpreta quel che c'è scritto nel *file* `java` che usa il file `Espressione.class` come dato da elaborare.

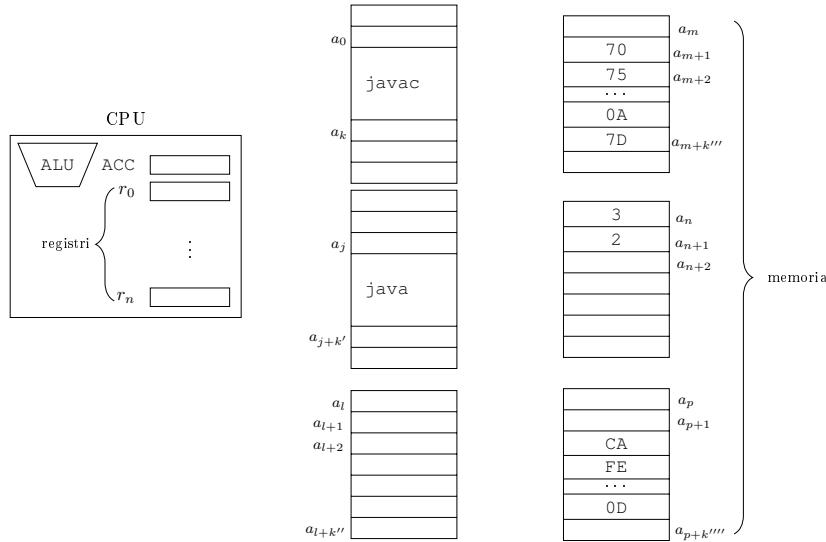
### 1.2.3 Interpretazione di `Espressione.class` a livello di CPU

Vediamo ora un'approssimazione di cosa significhi interpretare il codice oggetto `Espressione.class`. Non interpreteremo direttamente la sua rappresentazione esadecimale, che come ipotizzato, si trova a partire dall'indirizzo  $a_{p+2}$  della memoria. Interpreteremo, invece, la versione *disassemblata* di `Espressione.class` che abbiamo già visto:

```
Compiled from "Espressione.java"
public class Espressione {
 public Espressione();
 Code:
 0: aload_0
 1: invokespecial #1 // Method java/lang/Object."<init>":()V
 4: return

 public static void main(java.lang.String[]);
 Code:
 0: iconst_3
 1: istore_1
 2: iconst_2
 3: istore_2
 4: iload_1
 5: iload_1
 6: imul
 7: iload_2
 8: iload_1
 9: imul
 10: iadd
 11: istore_3
 12: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 15: iload_3
 16: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
 19: return
}
```

Partiamo con un'ipotesi semplificativa. Al termine dell'istruzione `istore_2` alla linea 3, supponiamo che lo stato della memoria sia come segue:



Ovvero, stabiliamo che i valori 3 e 2, assegnati ad  $a$  e  $b$  nel sorgente, siano “fisicamente” presenti agli indirizzi reali  $a_n$  e  $a_{n+1}$ , rispettivamente rappresentati dagli indirizzi  $_1$  e  $_2$  in `Espressione.class` disassemblato.

Le istruzioni seguenti serviranno per produrre il valore di  $r$  che potrebbe finire all’indirizzo  $a_{n+2}$ , ad esempio.

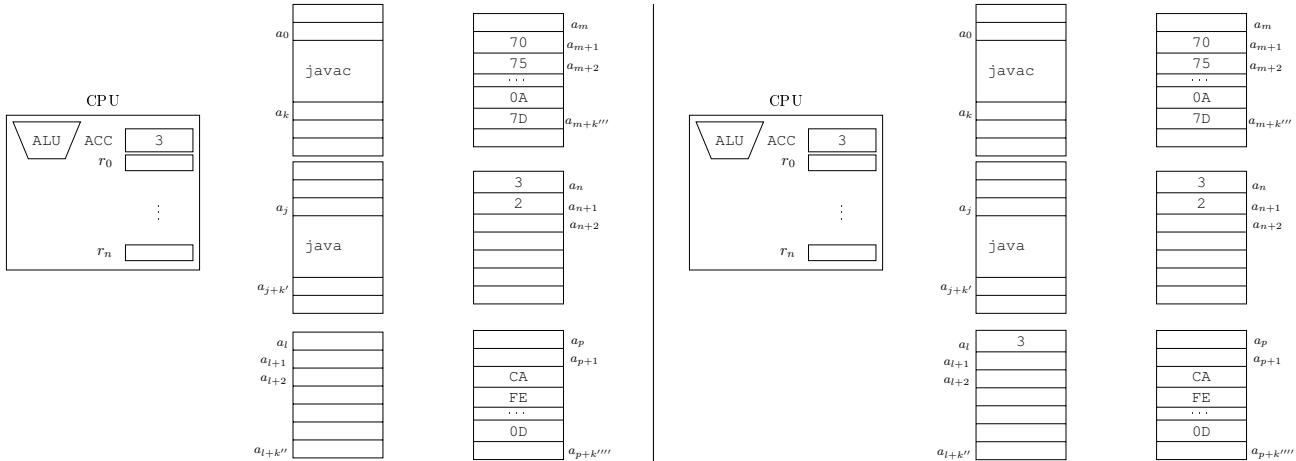
Vediamo ora come la CPU sia coinvolta per interpretare il blocco di istruzioni

```

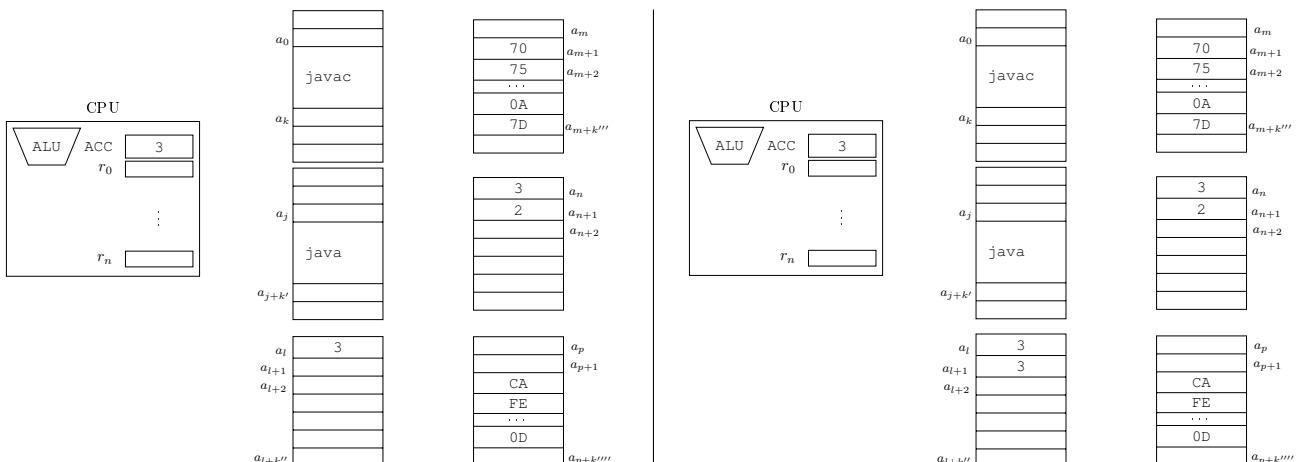
4: iload_1
5: iload_1
6: imult

```

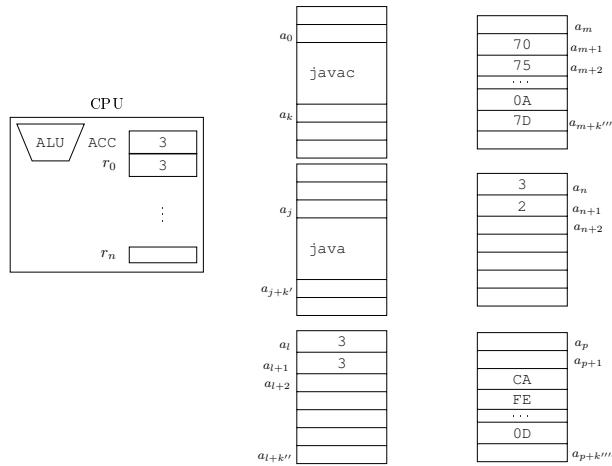
L’istruzione `iload_1` alla linea 4 indica che il valore 3 all’indirizzo  $a_n$  debba essere copiato in un’opportuna zona di memoria, l’*operand stack* che supponiamo inizi all’indirizzo fisico  $a_l$ . L’intero processo di copia richiede che 3 sia prima copiato in ACC, per poi essere ulteriormente copiato in  $a_l$ , come illustrato nella sequenza di “istantanee” qui sotto riportate:



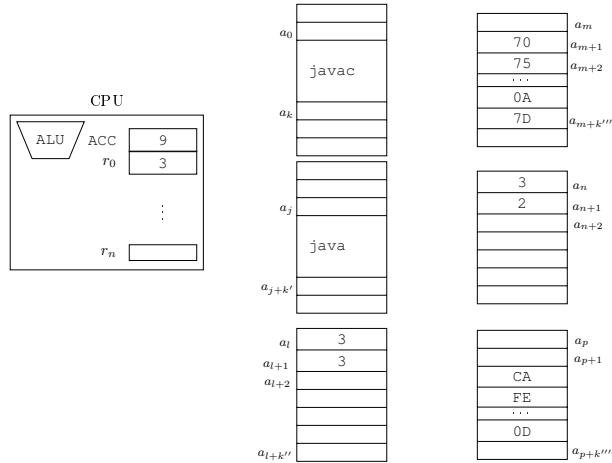
Siccome l’istruzione alla linea 5 è analoga alla precedente, i passi compiuti sono i seguenti:



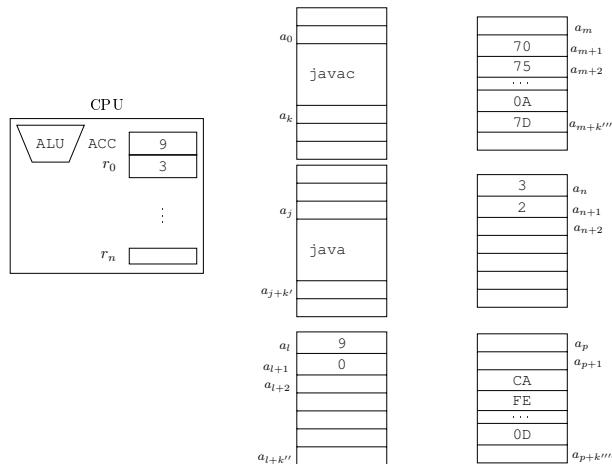
Siamo ora nella condizione di poter descrivere la sequenza di passi che realizzano l'interpretazione dell'istruzione `imult` a riga 6. La prima fase copia i due argomenti della moltiplicazione in ACC e nel primo registro utilizzabile, che supponiamo sia  $r_0$ . Siccome `imult` non contiene altra informazione, è sottinteso che i due suoi argomenti siano i primi valori disponibili nell'*operand stack*. La situazione diventa:



Una volta disponibili i valori in ACC e  $r_0$  la ALU effettivamente la moltiplicazione, sovrascrivendo l'attuale contenuto di ACC col risultato:



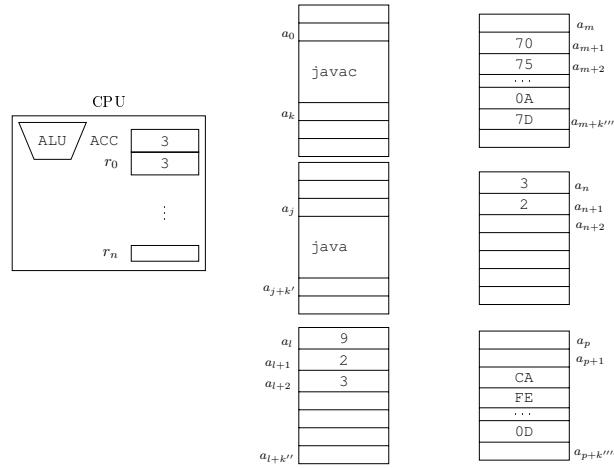
L'ultima fase elimina dall'*operand stack* i due argomenti usati per l'ultima operazione ed usa la prima cella libera dell'*operand stack* per memorizzare il contenuto di ACC:



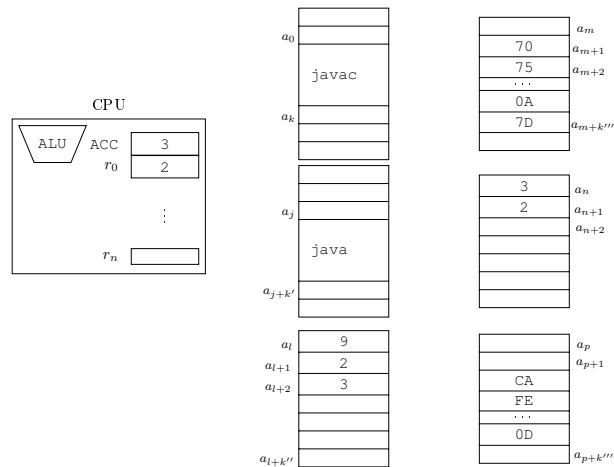
L'interpretazione di `Espressione.class` prosegue con:

```
7: iload_2
8: iload_1
9: imult
```

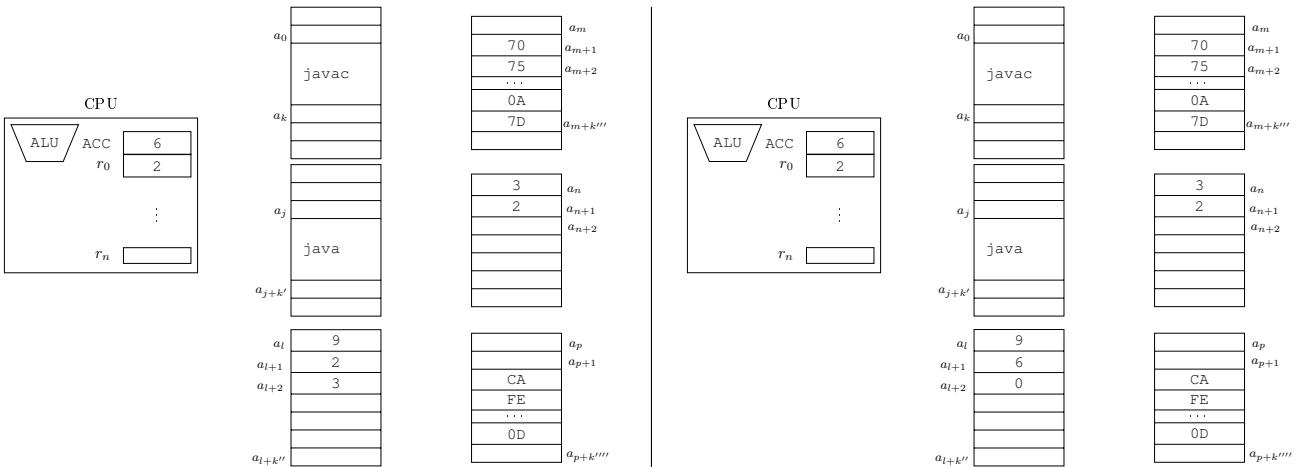
Al termine delle istruzioni alle righe 7 e 8 la situazione è simile a quella seguente le righe 4 e 5, avendo, però, tenuto conto del fatto che il valore intermedio 9 nell'*operand stack* non debba essere distrutto:



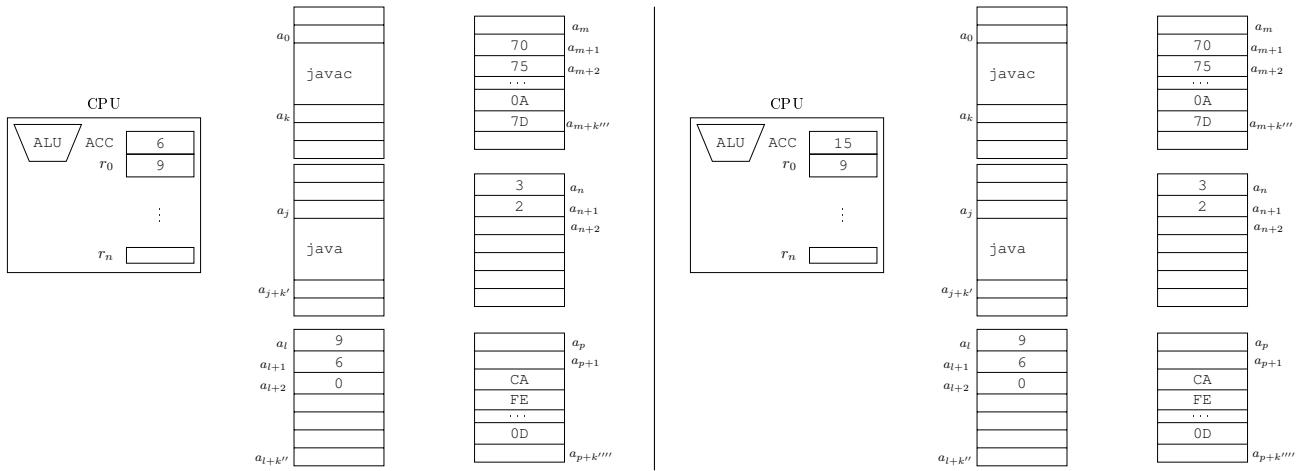
Dovendo reinterpretare una ulteriore occorrenza di `imult` il meccanismo è identico al precedente: i due argomenti in cima all'*operand stack* finiscono in ACC e  $r_0$ :



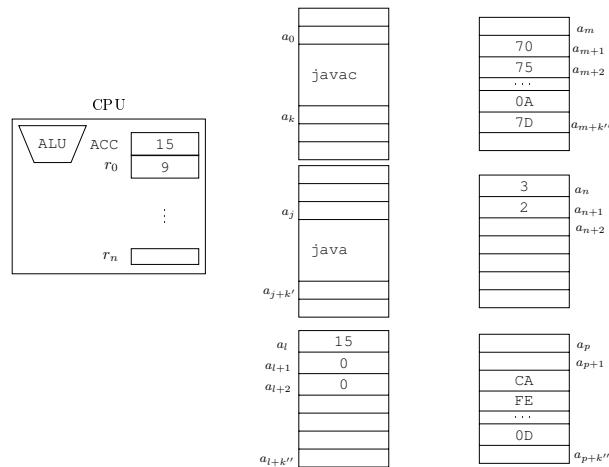
La ALU calcola la moltiplicazione, memorizzando il risultato in ACC, il quale, viene posto in cima all'*operand stack*:



Segue l'interpretazione di `iadd` che procede in perfetta analogia con quella di `imult`. I valori dei due argomenti necessari sono prima prelevati dalla cima dell'*operand stack* e copiati in ACC e  $r_0$  affinché la ALU calcoli il risultato, memorizzandolo in ACC:



Il contenuto di ACC è, infine memorizzato in cima all'*operand stack*, dopo aver eliminato i valori già usati:



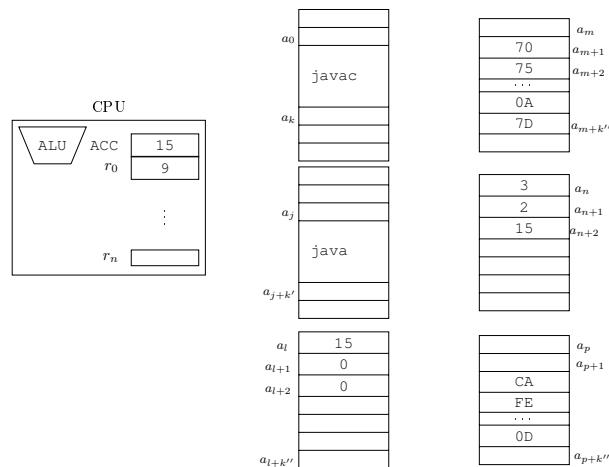
Rimangono le istruzioni:

```

11: istore_3
12: getstatic #2
15: iload_3
16: invokevirtual #3

```

Il dettaglio di `istore_3` è speculare a `iload_3`, memorizzando l'ultimo valore scritto nell'*operand stack* all'interno della cella di memoria destinata a contenere il valore della variabile `r`. Nel nostro caso supponiamo sia  $a_{n+2}$ :



Le istruzioni rimanenti, servono ad eseguire la pubblicazione del valore contenuto in  $a_{n+2}$ , ovvero del valore di `r`, sullo schermo.

**Conclusioni.** Dall'esempio appena sviluppato, dovrebbe essere evidente l'utilità dell'esistenza di linguaggi ad alto livello come, ad esempio, Java. In esso possiamo codificare algoritmi ponendoci praticamente allo stesso livello d'astrazione usato per descriverli. Un'enormità di dettagli che servono per distribuire coerentemente i dati in memoria e per manipolarli, rimangono nascosti grazie alla scomposizione del processo di codifica nelle due fasi descritte:

- scrittura di un sorgente che contenga il programma nella corretta sintassi del linguaggio di programmazione, nel nostro caso Java,
- compilazione del sorgente in oggetto che contiene la rappresentazione del sorgente adatta per essere interpretata, più o meno direttamente, dalla CPU.

La fase rilevante è quella a monte, ovvero di progettazione di algoritmi che risolvano in maniera generale un dato problema (computazionale) interessante.

**Esercizio 54 (Prime compilazioni.)** Svolgere l'esercizio descritto dal commento incluso nel sorgente delle seguenti classi Java

- [EsercizioUno.java](#),
- [EsercizioDue.java](#),
- [EsercizioTre.java](#),
- [EsercizioQuattro.java](#),
- [EsercizioCinque.java](#).

#### 1.2.4 Letture integrative

- Il [Wir73, Capitolo 3] illustra ed inquadra le parole chiave: *store, processing unit, registers, storage cells, unique address, binary encoding, finite range of values, machine code, stored-program computer, operation code, sharing the store between data and programs*.

Analoghi contenuti sono in [SM14, Capitolo 1], Sezione 1.1, Sezione 1.1.1, Sezione 1.1.2, Sezione 1.1.3.

- Riferimenti ai concetti di compilazione/compilatore, interpretazione/interprete, “*high-level*” *programming languages*, livelli di macchine virtuali sono in [Wir73, Capitolo 4] e [SM14, Capitolo 1], Sezione 1.1.4 .

# Capitolo 2

## Il linguaggio JAVA

### 2.1 Utilizzo essenziale delle classi

Trasformare un algoritmo in programma scritto in un qualche linguaggio di programmazione richiede la descrizione di una certa quantità di dettagli che dipendono dalla sia dalla sintassi del linguaggio scelto, sia dalla sua tecnologia implementativa.

Avendo noi l'obiettivo di usare Java come linguaggio di programmazione, guida irrinunciabile per fissare i dettagli è costituita almeno da una delle innumerevoli guide disponibili *on-line* sul linguaggio, o dal libro di testo di riferimento [SM14]. La differenza fondamentale tra una tipica guida *on-line*, quella ufficiale [The Java™ Tutorials](#) inclusa, e il testo [SM14] sta nel modo in cui il linguaggio viene usato. Il testo imposta l'uso del linguaggio in maniera classica, ignorando, per quanto possibile, il fatto che Java sia un linguaggio orientato alla progettazione ad oggetti.

I capitoli 2, 3, e 4 de [SM14] sono quelli di riferimento per impraticarsi all'uso del linguaggio in accordo con gli obiettivi di questo corso.

#### 2.1.1 Codifica di primi algoritmi in sorgenti Java

È essenziale rimarcare che, per ora:

- ogni sorgente Java che scriviamo si apre con una intestazione della forma:

```
1 class <Nome-classe> {
2 public static main (String[] args) {
3 // Corpo del programma che codifica un algoritmo
4 }
}
```

Il testo indica che il sorgente contiene una *classe* identificata col nome <Nome-classe>. Questo impone che il sorgente debba essere memorizzato nel *file system* <Nome-classe>.java, altrimenti la compilazione fallisce.

La parte di programma:

```
1 public static main (String[] args) {
2 // Corpo del programma che codifica un algoritmo
3 }
```

descrive il *metodo principale* della classe. Esso ha nome main. Costituisce il punto di ingresso dell'interprete java, ovvero individua il punto da cui comincia l'interpretazione del *file* oggetto di nome <Nome-classe>.class.

- l'unica interazione che avremo con i programmi sarà la stampa dell'eventuale risultato prodotto. La stampa avviene *richiamando* il metodo:

```
1 System.out.println(<argomento>);
```

oppure il metodo:

```
1 System.out.print(<argomento>);
```

L'`<argomento>` può esser sintatticamente molto complesso. L'idea è di imparare con la pratica il minimo indispensabile per rendere comprensibile quanto viene stampato sullo standard output, che, generalmente, è proprio lo schermo del PC sui cui si lavora. A tal fine, costituiscono riferimento il testo e gli esempi di programmi. La differenza tra i due metodi è che il primo, dopo la pubblicazione dell'argomento porta il cursore sulla linea successiva, mentre il secondo lo lascia sulla stessa.

## 2.2 Tipi di base

La descrizione dell'insieme di valori che una variabile possa memorizzare e, quindi, delle operazioni che su di essa siano ammissibili, è uno dei dettagli fondamentali della progettazione di un programma.

Se ben giustificate, tali informazioni aiutano sia il compilatore nel determinare scelte poco coerenti nella manipolazione di dati sia nel *debugging* di un programma, ovvero nel processo di eliminazione degli errori, che possono esistere anche se il programma è sintatticamente corretto.

- [Numeri interi](#),
- [Tipi primitivi](#),
- [ValoriMinimiMassimiInteri.java](#),
- [Overflow.java](#),
- [ValoriMinimiMassimiNonInteri.java](#).

## 2.3 Ambiguità sintattica della selezione

- [ElseNoParentesi.java](#),
- [ElseNoIf.java](#),
- [IfAmbiguita.java](#).

## 2.4 Operatori ed espressioni Java

[Operator summary](#) di Oracle ed alcuni esempi relativi ad operatori ed espressioni booleane:

- [OperatoriBooleani.java](#);
- [EspressioniBooleane.java](#);
- [LazyAnd.java](#), [LazyOr.java](#).

## 2.5 Programmazione e codifica: stili

Illustrazione degli [Indent style](#).

## 2.6 Sviluppo di codice con classi e metodi

La classe `java.lang.Math` de [Oracle Javadocs API](#). fornisce metodi per l'esecuzione di operazioni numeriche di base come l'esponenziale, il logaritmo, la radice quadrata, e le funzioni trigonometriche, ad esempio. Tipicamente, i metodi di `java.lang.Math` coincidono con quelli della classe `java.lang.StrictMath` che fornisce implementazioni di algoritmi per funzioni numeriche in grado di produrre risultati di qualità paragonabile a quelli prodotti dai programmi della "Freely Distributable Math Library" [netlib](#), raccolta di implementazioni in C di algoritmi per funzioni quali, ad esempio, sin, cos, tan, asin, acos, atan, exp, log, log10, ....

### Costanti fondamentali

La classe `java.lang.Math` contiene i campi statici `final` `Math.E`, che mette a disposizione il valore del [Numero di Eulero o Nepero](#) e `Math.PI`, col valore di  $\pi$ .

## Metodi interessanti

Ad esempio, il metodo statico `int Math.addExact(int, int)` in `java.lang.Math` restituisce la somma dei suoi argomenti, segnalando l'errore che ci si aspetta di ottenere in caso di *overflow*. Esistono metodi analoghi per altre operazioni aritmetiche di base, ovviamente.

### 2.6.1 Classe con metodi iterativi per l'aritmetica

Scopo di questa sezione è sollecitare la programmazione, o implementazione, di una classe che contenga metodi che realizzino gli algoritmi più volte discussi in aula per operazioni aritmetiche su numeri naturali.

`LibreriaAritmeticaBozza.java` imposta la struttura della una classe.

Implementando la classe, sarà opportuno attenersi al seguente vincolo: i metodi per operazioni più complesse devono richiamare metodi che implmentino funzioni meno complesse. Ad esempio, il metodo `molt(int, int)` che implementi la moltiplicazione, dovrà sfruttare il metodo `piu(int, int)` che implementi la somma, già disponibile nella classe in via di definizione.

Oltre alla classe principale, con i metodi per le operazioni aritmetiche, definire

`LibreriaAritmeticaTestBozza.java` con il metodo `main` ed eventuali ulteriori metodi il cui scopo sia il *testing* dei metodi in `LibreriaAritmeticaBozza.java`. È, infatti, opportuno separare la progettazione di classi in quelle di libreria e quelle di *testing*.

(`LibreriaAritmetica.java` è una possibile implementazione di `LibreriaAritmeticaBozza.java`.)

**Esercizio 55 (Classe con metodi e classe di collaudo)** Estendere `LibreriaAritmeticaBozza.java`, e `LibreriaAritmeticaTestBozza.java` con:

- metodi che risolvano i problemi PDSP, SPNP e MSMP dell'Esercizio 10.

Le classi `PariDispariIterativo.java`, `Accumulatore.java` e `MinimoInParallelo.java` raccolgono e discutono brevemente possibili soluzioni;

- un metodo che calcoli il fattoriale;

La classe `Fattoriale.java` contiene una implementazione del fattoriale;

- un metodo che risolva il problema QPNP della Sottosezione 2.3.1.

(`QuadratoConRaddoppiEtc.java` è una possibile soluzione.)

■

**Esercizio 56 (Classe con metodi ricorsivi per l'aritmetica.)** 1. Definire una classe in cui ogni metodo sia ricorsivo co-variante e calcoli la stessa funzione di uno dei metodi in `LibreriaAritmetica.java`.

2. Scrivere una classe analoga alla precedente, i cui metodi ricorsivi siano contro-varianti.

3. Scrivere una classe analoga alla precedente, i cui metodi ricorsivi siano di coda o dicotomici, quando possibile.

4. Alle classi precedenti, con metodi che implementino le operazioni aritmetiche iterativamente e ricorsivamente, aggiungere un metodo iterativo o ricorsivo che stampi la tavola pitagorica.

(`PitagoricaRec.java` contiene un possibile suggerimento.)

■

## 2.7 Input da tastiera

La classe `SIn.java` documentata in `SIn Javadoc` per la `LetturaDiEnneInteri.java` in cui si usano anche stringhe per comunicare con l'utente.

**Esercizio 57 (Letture di sequenze di valori da tastiera)** Scrivere metodi iterativi, ricorsivi co/contro-varianti e dicotomici che realizzino le seguenti operazioni di filtro su valori letti da tastiera. Quando possibile, usare metodi di classi già definite.

1. Fissato un valore  $n \geq 1$ , leggere  $n$  interi, ma stampare solo i dispari.
2. Fissati due valori  $n \geq 1$ , e  $s$ , leggere  $n$  interi, ma stampare solo quelli maggiori di  $s$ .
3. Fissati due valori  $n \geq 1$ , e  $s$ , leggere  $n$  interi, ma stampare solo quelli che siano il doppio di  $s$ .
4. Fissati tre valori  $n \geq 1$ ,  $m$  ed  $M$ , leggere  $n$  interi, ma stampare solo quelli minori di  $m$  o maggiori di  $M$ .
5. Fissati tre valori  $n \geq 1$ ,  $m$  ed  $M$ , leggere  $n$  interi, ma stampare solo quelli nell'intervallo  $[m, M]$ .
6. Fissato un valore  $n \geq 1$ , leggere  $n$  interi, e calcolarne la media.

## 2.8 Tipi numerici non interi

**Esercizio 58 (Numero di Eulero)** Scrivere una classe con metodi opportuni per calcolare sia il [Numero di Eulero](#) o [Nepero](#), usando la definizione di  $e$  come serie numerica, sia di  [\$\pi\$](#) , usando le formule di Viète, Leibniz, Nilakantha, Madhava, Wallis, il prodotto infinito di Eulero con i numeri primi dispari, la serie armonica, etc. .

Fare il *testing* della classe, definendo una opportuna classe di *testing*. ■

**Esercizio 59 (Una classe per il calcolo combinatorio.)** Definire le seguenti tre classi:

- CombinatoriaIter con metodi iterativi che calcolino il valore di un coefficiente binomiale ed il numero di permutazioni e disposizioni per un certo insieme;
- CombinatoriaRic con metodi ricorsivi che calcolino il valore di un coefficiente binomiale ed il numero di permutazioni e disposizioni per un certo insieme;
- Tartaglia con metodi iterativi e ricorsivi che stampino su standard output il Triangolo di Tartaglia.

Un punto di partenza per definire i metodi necessari possono essere informazioni reperibili on-line, oppure, per il calcolo delle disposizioni, ci si può rifare alla seguente definizione ricorsiva:

$$d(n, k) = \begin{cases} k! & \text{se } n = k \\ n/(n - k) * d(n - 1, k) & \text{se } n > k \end{cases} .$$

È, ovviamente, obbligatorio separare le classi CombinatoriaIter, CombinatoriaRic e Tartaglia da quelle di *testing*. ■

## 2.9 Operazioni su array

**Esercizio 60 (Filtri su array)**

Definire una classe che contenga metodi adatti a risolvere i problemi qui sotto elencati. Il suggerimento è sintetizzare più metodi per uno stesso problema, impostandone di iterativi e di ricorsivi.

Riguardo ai metodi iterativi, vale la pena differenziarli per mezzo di scelte opportune su come generare l'*array* che contiene il risultato. In generale, scrivere un metodo `int[] filtroInvolutro(int[])` richiama il filtro vero e proprio cui è demandato il compito di generare l'*array* risultato. In alternativa, l'*involutro* può generare l'*array* che verrà modificato dal filtro, sfruttando l'*aliasing*.

Riguardo ai metodi ricorsivi, oltre all'opportuna gestione dell'*array* risultato attraverso un metodo *involutro*, potranno essere co/contro-varianti o dicotomici:

- scrivere il metodo `int[] initArray()` che, letto un intero  $n > 0$  da tastiera, restituisca un *array* con  $n$  valori interi, anch'essi letti da tastiera;
- `int[] filtroMinoriDi(int[] a, int limiteSuperiore)` che restituisca l'*array* di interi copiati da *a* che siano minori del valore *limiteSuperiore*;
- `int[] filtroMaggioriDi(int[] a, int limiteInferiore)` che restituisca un *array* di interi copiati da *a* che siano maggiori del valore *limiteInferiore*;
- `int[] filtroDispari(int[] a)` che restituisca un *array* di interi dispari copiati da *a*;
- `int[] filtroIntervallo(int[] a, int min, int max)` che restituisca un *array* di interi copiati da *a* compresi tra *min* e *max*;
- `int[] filtroDispariDoppioDi(int[] a, int riferimento)` che restituisca un *array* di interi copiati da *a* che siano il doppio del valore in *riferimento*;
- `boolean[] trasduttore(int[] a, int limiteSuperiore)` che restituisce un *array* di booleani, in cui ogni elemento sia `true` se l'elemento di posizione corrispondente in *a* è inferiore a *limiteSuperiore* e `false` altrimenti;
- `int indiceSubSeq(int[] a, int[] b)` che restituisce l'indice al primo elemento della prima istanza della sotto-seguenza non vuota *b* in *a*. Ad esempio, se *x* = {1, 2, 3, 1, 2, 3} e *y* = {2, 3}, allora `indiceSubSeq(x, y)` restituisce 1. Se la sotto-seguenza non esiste, il risultato è, per convenzione, -1.

## 2.10 Il tipo char

[TipoChar.java](#) è una classe riassuntiva essenziale che aiuta a focalizzare quali valori contenga il tipo base char.

**Esercizio 61 (Menù di crescente difficoltà)** 1. Scrivere una classe che offra il seguente menù a molteplici voci, offrendo all'utente di compiere una scelta:

- a. New document
- b. Open document
- c. Save as ...
- d. Mail document to ...

Please, make your choice (a-d) :

Prima di terminare, in base alla scelta, la classe deve stampare sullo standard out una delle seguenti opzioni:

**Scelta a)** You chose to create a new document.

**Scelta b)** You chose to open a document.

**Scelta c)** You chose to save the document.

**Scelta d)** You chose to send the document.

Per ogni scelta al di fuori dell'intervallo il messaggio deve essere Your choice is not valid..

([MenuBase.java](#) è una possibile soluzione.)

2. Estendere il menù all'esercizio precedente, con una ulteriore voce finale:

- e. Quit

e modificarne il comportamento in modo che, l'offerta della scelta tra le opzioni continui finché l'utente non scelga proprio l'opzione e.

([MenuConCiclo.java](#) è una possibile soluzione.)

**Esercizio 62 (Menù per calcolatrice)** Scrivere una classe menù adatta a realizzare una semplice calcolatrice, la quale usi i metodi disponibili della classe libreria [LibreriaAritmeticaBozza.java](#), sviluppata nell'Esercizio 55.

## 2.11 La classe String

Riferimento per questo argomento è il [SM14, Capitolo 6] Sezione 6.2. Le istanze della classe String sono array di elementi con tipo primitivo char.

[StringIntroduzione.java](#) è una classe essenziale per evidenziare quel che della documentazione on-line su [String](#) vale la pena approfondire: la concatenazione di stringhe, i metodi **non statici** (di istanza) length, equals, charAt, substring.

Invece:

- [ArgsMain.java](#) evidenzia il possibile uso del parametro formale String[] del metodo main e
- [ArrayDiString.java](#) è un esempio essenziale di classe che permette di inizializzare un *array* di tipo String.

**Esercizio 63** 1. Scrivere un metodo String `toString(int[] a)` che legga tutti i valori in a e li restituisca concatenati in una stringa. La stringa li deve elencare, separati da ",". Ad esempio, se a è definita come {1, 2, 3} la stringa risultante sarà "1, 2, 3".

2. Scrivere un metodo String `toStringPuntoFinale(int[] a)` che legga tutti i valori in a e li restituisca concatenati in una stringa. La stringa li deve elencare, separati da "," e deve concludersi con ". ". Ad esempio, se a è definita come {1, 2, 3} la stringa risultante sarà "1, 2, 3.". "

3. Scrivere un metodo statico `substring (String s, int i, int f)` che si comporti come quello (di istanza) `substring` della classe String fornita da Java e che usi solo il metodo (di istanza) `charAt`.

4. Scrivere un metodo statico `equals (String a, String b)` che si comporti come quello (di istanza) `equals` della classe String fornita da Java e che usi solo i metodi (di istanza) `charAt` e `length`.

5. Scrivere due metodi, uno iterativo, l'altro ricorsivo che, letta una stringa da tastiera, determini se essa sia palindroma, ovvero se essa possa essere letta da sinistra a destra e da destra a sinistra con lo stesso risultato. Esempi di stringhe palindrome sono radar, anna, otto. Per leggere una stringa tramite tastiera è possibile usare l'istruzione:

```
1 String s = SIn.readLine();
```

[LibreriaCreazioneStampaArray.java](#) fornisce soluzioni possibili alle richieste 1 e 2.

[StringMetodi.java](#) fornisce soluzioni possibili alle richieste 3 e 4.

[StringaPalindroma.java](#) fornisce una soluzione possibile all'ultimo punto.

6. Scrivere due metodi, uno iterativo, l'altro ricorsivo che, letta una stringa da tastiera, restituisca la stringa rovesciata. ■

## 2.12 Array multidimensionali

Svolgere i seguenti esercizi, combinando, per ciascun punto elencato, le diverse strategie di visita di righe e colonne. Ad esempio, si intende che, svolgendo il primo punto dell'Esercizio 64, le righe della matrice multidimensionale sono percorse ricorsivamente, in uno dei modi conosciuti, mentre le colonne, sono visitate iterativamente. Ovviamente anche la combinazione opposta o l'uso della sola ricorsione, o iterazione, per la visita di entrambe le dimensioni disponibili sono altrettanti esercizi utili.

**Esercizio 64** Sia dato l'*array*  $a = \{12, 21, 33, 45, 73\}$  di interi.

1. Definire un metodo *init* che produce una matrice  $m$  di cinque righe in cui le righe di indice pari sono una copia di  $a$  e quelle dispari sono null.
2. Scrivere un metodo *linRighe* che linearizza  $m$  costruita al punto 1 in un *array* sufficientemente capiente, percorrendo  $m$  *riga* per *riga*.
3. Scrivere un metodo *linColonna* che linearizza  $m$  costruita al punto 1 in un *array* sufficientemente capiente, percorrendo  $m$  *colonna* per *colonna*.
4. Scrivere un metodo *diagI* che costruisce una matrice  $n$  come la seguente:

```
{ { 12, 12, 12, 12, 12}
, { 12, 21, 21, 21, 21}
, { 12, 21, 33, 33, 33}
, { 12, 21, 33, 45, 45}
, { 12, 21, 33, 45, 73} }
```

usando l'*array*  $a$  come sorgente degli elementi nelle diagonali principali.

5. Scrivere un metodo *diagF* che costruisce una matrice  $n$  come la seguente:

```
{ { 12, 21, 33, 45, 73}
, { 21, 21, 33, 45, 73}
, { 33, 33, 33, 45, 73}
, { 45, 45, 45, 45, 73}
, { 73, 73, 73, 73, 73} }
```

usando l'*array*  $a$  come sorgente degli elementi nelle diagonali principali. ■

**Esercizio 65** Sia  $m$  una matrice multidimensionale di interi, possibilmente *ragged*. Scrivere un metodo opportuno per ciascuno dei seguenti punti.

1. Calcolare la somma di tutti gli elementi di  $m$ .
2. Produrre un *array* in cui ogni elemento contiene la somma degli elementi di ciascuna riga in  $m$ .
3. Produrre un *array* in cui ogni elemento contiene la somma degli elementi di ciascuna colonna in  $m$ .
4. Produrre un *array* in cui ogni elemento contiene la somma degli elementi di ciascuna diagonale principale in  $m$ .
5. Produrre un *array* in cui ogni elemento contiene la somma degli elementi di ciascuna diagonale secondaria in  $m$ .

6. Produrre un *array* in cui ogni elemento contiene la differenza tra la somma di tutti gli elementi della  $2n$ -esima *riga* e la somma della  $2n + 1$ -esima *riga* in  $m$ .
7. Produrre un *array* in cui ogni elemento contiene la differenza tra la somma di tutti gli elementi della  $2n$ -esima *colonna* e la somma della  $2n + 1$ -esima *colonna* in  $m$ .
8. Produrre un *array* in cui ogni elemento contiene la differenza tra la somma di tutti gli elementi della  $2n$ -esima *riga* e la somma della  $2n + 1$ -esima *colonna* in  $m$ . Tenere conto del fatto che  $m$  non è necessariamente una matrice quadrata.
9. Produrre la media di tutti gli elementi di  $m$  che superano una soglia data  $v$ . Ripetere il calcolo relativamente ai soli elementi che si trovino o sulla diagonale maggiore o su quelle superiori. ■

Completare l'esercizio, definendo anche classi di *testing* opportune per ogni metodo scritto.

## 2.13 Problemi decisionali su *array*

**Esercizio 66** Risolvere iterativamente i seguenti problemi, seguendo l'impostazione suggerita alla Sottosezione 6.7.1, della prima parte delle dispense, anche se i parametri dei metodi non sono matrici bidimensionali, ma "solo" *array*:

1. Sia  $a$  un *array* di interi. Scrivere un metodo iterativo *almenoDue* che restituisce *true* se in  $a$  esiste almeno un elemento  $a[i]$  per cui si possa verificare che ogni altro elemento  $a[j]$  di  $a$  sia tale che il valore assoluto della differenza  $|a[i]-a[j]|$  vale almeno due. Deve restituire *false* in ogni altro caso.
2. Sia  $a$  un *array* di interi. Scrivere un metodo *multiploDx* che restituisce *true* se in  $a$  esiste almeno un elemento che è multiplo di tutti gli elementi alla sua destra in  $a$ . Deve restituire *false* in ogni altro caso.
3. Siano  $a$  e  $b$  due *array* di interi. Scrivere un metodo *sommaDueCons* che restituisce *true* se per ogni elemento di  $a$  esistono due elementi consecutivi in  $b$  la cui somma è pari ad  $a$ . Deve restituire *false* in ogni altro caso.

Completare l'esercizio, definendo anche classi di *testing* opportune per ogni metodo scritto.

La classe [Quantificatori.java](#) fornisce possibili soluzioni iterative. ■

**Esercizio 67** Riprogettare ricorsivamente le soluzioni date per l'Esercizio 66. ■

## 2.14 Eventuali approfondimenti (NON parte del programma didattico)

### 2.14.1 API Java

[Java™ Platform, Standard Edition 7 API Specification](#) fornisce la documentazione di package e classi in essi contenuti.

Nel *package* `java.lang` troviamo quelle sia quelle corrispondenti a tipi di base come, ad esempio `Boolean` e `Integer`, sia altre di utilità come `Math`. Un altro *package* che vale la pena di esplorare è `java.util` in cui esistono le classi `Arrays` e `Scanner`.

### 2.14.2 Package

La [SM14, Sezione 9.8] descrive cosa siano i *package*.

L'archivio [ScompattareInUnaSolaCartella.zip](#) permette di capire come organizzare un semplice *package* e di compilarlo, seguendo la traccia data in [Java Package](#) che non richiede necessariamente la configurazione della variabile `CLASSPATH` di sistema. In particolare, una volta posizionati nella cartella che contenga le cartelle `packages` e `classi`, i comandi da impartire per compilare ed utilizzare `LibreriaAritmeticaTest` sono:

- in ambienti Unix:

```
javac packages/aritmetica/LibreriaAritmetica.java classi/LibreriaAritmeticaTest.java
java classi.LibreriaAritmeticaTest
```

- in ambienti DOS/Windows:

```
javac packages\aritmetica\LibreriaAritmetica.java classi\LibreriaAritmeticaTest.java
java classi.LibreriaAritmeticaTest
```

**Esercizio 68 (Un *package* con le costanti fondamentali.)** Definire una classe `CostantiNumericheFondamentali` in cui esistano più metodi per calcolare il numero di Eulero o  $\pi$ . Includere `CostantiNumericheFondamentali` nel *package* `packages.aritmetica` e aggiornare `classi.LibreriaAritmeticaTest` per verificare il comportamento dei metodi in `CostantiNumericheFondamentali`.

## 2.15 Assert in Java

Java mette a disposizione le direttive `assert`, utilizzabili per verificare in maniera più sistematica la correttezza parziale dei programmi: [Using Assertions in Java Technology](#) e [Programming with Assertions in Java](#).

Programmi già sviluppati, corredati di istanze di `assertion`, applicate ad invarianti di ciclo e predicati di uscita dai cicli per verificarne la correttezza parziale in maniera più sistematica, di quanto non sia possibile con batterie di casi di prova (*testing*):

- [SommaSuccessoreIterato.java](#),
- [MoltiplicazioneSommaIterata.java](#),
- [QuozienteDifferenzaIterata.java](#),

Anche [RestoDifferenzaIterata.java](#), è uno dei programmi già sviluppati.

Lo trattiamo separatamente dagli altri per la sua peculiarità. Esso evidenzia il seguente aspetto rilevantissimo:

L'introduzione di direttive `assert` in programmi Java permette una verifica più sistematica della presenza di errori, che, però, non è esaustiva.

Il sorgente [RestoDifferenzaIterata.java](#), suggerisce modifiche che superano la verifica dei predicati cui le `assert` sono applicate, anche se il programma modificato non è corretto.

Questo caso di studio è fondamentale per sottolineare che il processo di dimostrazione della verità di un predicato, che eventualmente formalizzi le proprietà di un programma, è un processo più sofisticato di quanto non sia la verifica per casi interessanti (*testing*), pur condotta in maniera sistematica.

### Esercizi proposti su `assert`.

1. Aggiungere le `assert` ragionevoli alle classi seguenti, già sviluppate o viste in precedenza:

[PotenzaMoltiplicazioneIterata.java](#), [DifferenzaPredecessoreIterato.java](#),  
[QuadratoProdottoNotevole.java](#).

(Soluzioni:

- [PotenzaMoltiplicazioneIterata.java](#),
- [DifferenzaPredecessoreIterato.java](#),
- [QuadratoProdottoNotevole.java](#),
- [ContaPariDispari.java](#).)

# Bibliografia

- [SM14] W. Savitch and D. Micucci. *Programmazione di base e avanzata con Java*. Pearson, 2014.
- [Wir73] Niklaus Wirth. *Systematic Programming: An Introduction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.