Dispensa di Programmazione

Introduzione alla Programmazione (a.a. 2022/23).

Giulio Caravagna

Corso di Laurea Triennale in Intelligenza Artificiale e Data Analytics (AIDA)
Departimento di Matematica e Geoscienze
Università di Trieste

28 settembre 2023

Indice

I Principi di programmazione imperativa **5**

1 Introduzione e cenni storici **6**
- 1.1 Storia della programmazione 6
- 1.2 Paradigmi di programmazione 7

2 Nozioni fondamentali **9**
- 2.1 Programma e computazione 9
- 2.2 Assegnamento ed espressioni 11
- 2.3 Ambiente e memoria 16
- 2.4 Dichiarare una variabile 17

3 Istruzioni condizionale ed iterative **21**
- 3.1 Istruzione condizionale `if` 21
- 3.2 Espressioni logiche 23
- 3.3 Istruzione iterativa `while` 27
- 3.4 Istruzione iterativa `for` 31
- 3.5 Blocchi di codice e pile di frame 32
- 3.6 Struttura di un programma in `C` 35

4 Funzioni **39**
- 4.1 Il concetto di funzione nella programmazione 39
- 4.2 Passaggio dei parametri, e semantica dei blocchi 40
- 4.3 Funzioni ricorsive 45
- 4.4 Funzioni iterative e ricorsive per il calcolo di serie e successioni 47

5 Puntatori **51**
- 5.1 Definizione e modello in memoria 51
- 5.2 Passaggio di parametri per indirizzo 54

6 Vettori **57**
- 6.1 Lettura e scrittura di valori in un array 58
- 6.2 Array, memoria e puntatori 61
- 6.3 Problem solving su array 64

6.4 Stringhe 73

7 Memoria dinamica 76
7.1 Allocazione della memoria 76

8 Liste linkate 82
8.1 Definizione di liste 83
8.2 Creazione di liste 85
8.3 Manipolazione di liste 88

II Esercizi 93

9 Sintassi di base 93

10 Funzioni iterative e ricorsive 99

11 Serie e successioni 107

12 Puntatori e Array 109

III Principi di programmazione ad oggetti 113

13 Introduzione a `Python` 113

14 Programmazione ad oggetti 114
14.1 Classi ed oggetti 114
14.2 Ereditarietà tra classi 117
14.3 Classi astratte 126
14.3.1 Il decoratore abstractmethod 131

15 Meccanismi avanzati di iterazione 132
15.1 List comprehensions 132
15.2 Iteratori 134
15.3 Liste concatenate in `Python` 140

IV Esercizi 142

16 Ereditarietà 142

17 Classi astratte 145

18 List comprehension 147

19 Iteratori 148

V Alcuni esercizi risolti e spiegati 150

20 Serie e successioni 150

21 Funzioni ricorsive 153

22 Vettori e algoritmi 157

23 Liste 160

Preambolo. Questa dispensa rappresenta il materiale didattico per gli studenti di "Introduzione alla Programmazione" (a.a. 2022/23)– ex "Programmazione e architetture degli elaboratori" – svolto presso la Laurea Triennale in Intelligenza Artificiale e Data Analytics (AIDA) e mutuato in altri corsi di laurea dell'Univesità di Trieste.

La dispensa introduce, partendo da zero, i principi base del pensiero computazionale attraverso la programmazione imperativa (in linguaggio `C`), fino ad arrivare ai principi della programmazione ad oggetti ed accennando anche alcuni principi di programmazione funzionale (in linguaggio `Python`).

Il corso – da 9 CFU – complementa un "Laboratorio di Programmazione" dedicato allo sviluppo software in `Python` (3 CFU). I corsi hanno obiettivi e materiali didattici diversi. ll corso di programmazione introduce ai concetti alla base della cosiddetto *computational thinking*, focalizzandosi su principi fondamentali - soprattuto di ragionamento - che valgono a prescindere dal linguaggio in cui si programma. Il Laboratorio insegna a programmare moduli `Python` in modo concreto, toccando argomenti specifici come gestione del codice con versionamento, unit tests e sanitizzazione. L'obiettivo di questo corso è quindi costruire una base solida e trasversale per programmare agilmente in diversi linguaggi di programmazione, permettendo allo studente di imparare nuovi linguaggi in breve tempo!

Guida alla dispensa. Questa dispensa si "evolve" di anno in anno; dubbi, chiarimenti ed eventuali errori possono essere segnalati via mail a gcaravagna@units.it.

Nel testo le spiegazioni sono colloquiali, sacrificando il rigore per la chiarezza espositiva per fornire un accesso semplice e diretto ai concetti base. Esempi di codice sono colorati e discussi sia in `C` che `Python`, linguaggi presi ad esempio per praticità, popolarità e rilevanza. In generale una variabile indicata come x, y, etc. denota una quantità matematica, mentre `x` o `y` una variabile scritta nel programma, quindi codice. Spesso le due coincidono e la notazione non è necessariamente consistente tra le varie sezioni.

Ringraziamenti. La versione iniziale di questa dispensa è stata preparata da Gaia Saveri sbobinando le lezioni dell'a.a. 2021/22. Ulteriori ringraziamenti vanno a Ciro Antonio Mami, Pietro Morichetti, assistenti alla didattica per l'a.a. 2020/21 e 2021/22, ed a Vittorio Amoruso, studente per l'a.a. 2020/21.

Principi di programmazione imperativa

PART

I

Sezione 2. Introduzione
Sezione 2.2. Assegnamento ed espressioni in C
Sezione 2.3. Rappresentazione di ambiente e memoria
Sezione 2.4. Dichiarazioni di variabili in C

1 Introduzione e cenni storici

1.1 La Storia della Programmazione Informatica e l'Evolvere del Pensiero Computazionale [secondo ChatGPT]

La storia della programmazione informatica è un affascinante viaggio attraverso le epoche, le idee e le innovazioni che hanno trasformato il modo in cui interagiamo con le macchine e risolviamo i problemi. Questo breve saggio esplorerà le tappe fondamentali di questa storia, concentrandosi sui principali paradigmi di programmazione - imperativo, ad oggetti e funzionale - e sullo sviluppo del pensiero computazionale.

I Primi Passi: Il Paradigma Imperativo La programmazione informatica ha le sue radici nella metà del XX secolo, quando i primi computer elettronici iniziarono a emergere. All'inizio, la programmazione era dominata dal paradigma imperativo, che si basa sulla sequenza di istruzioni eseguite dalla macchina. Uno dei linguaggi più noti di questa epoca è il Fortran, sviluppato negli anni '50 da IBM. Fortran fu uno dei primi linguaggi di alto livello e fu ampiamente utilizzato nella simulazione scientifica e nell'analisi numerica.

L'Età dell'Oggetto: Il Paradigma Ad Oggetti Negli anni '60 e '70, la programmazione informatica ha subito una trasformazione significativa con l'introduzione del paradigma ad oggetti. Questo approccio ha introdotto il concetto di oggetto come entità che racchiude dati e metodi, consentendo una migliore organizzazione e astrazione del codice. Il linguaggio di programmazione Simula, sviluppato da Ole-Johan Dahl e Kristen Nygaard in Norvegia, è stato uno dei precursori del paradigma ad oggetti. Tuttavia, è con l'avvento di linguaggi come C++ e Java che questo paradigma ha guadagnato popolarità su vasta scala.

Il paradigma ad oggetti ha rivoluzionato lo sviluppo software, rendendo possibile la modellizzazione dei problemi reali in modo più naturale ed efficiente. I concetti di incapsulamento, ereditarietà e polimorfismo sono diventati pilastri della programmazione orientata agli oggetti (OOP). Questo approccio ha reso più facile la gestione della complessità del software, favorendo la riusabilità del codice e migliorando la manutenibilità dei sistemi.

La Rivoluzione Funzionale: Il Paradigma Funzionale Negli anni '70, mentre il paradigma ad oggetti stava prendendo piede, un altro approccio stava emergendo: il paradigma funzionale. La programmazione funzionale si basa sulla valutazione di funzioni matematiche e minimizza l'utilizzo di variabili e stati. Lisp, uno dei primi linguaggi di programmazione funzionale, è stato sviluppato da John McCarthy ed è diventato uno strumento essenziale per l'intelligenza artificiale.

Con l'evolversi del paradigma funzionale, linguaggi come Haskell e OCaml hanno portato l'attenzione sulla programmazione dichiarativa, dove il pro-

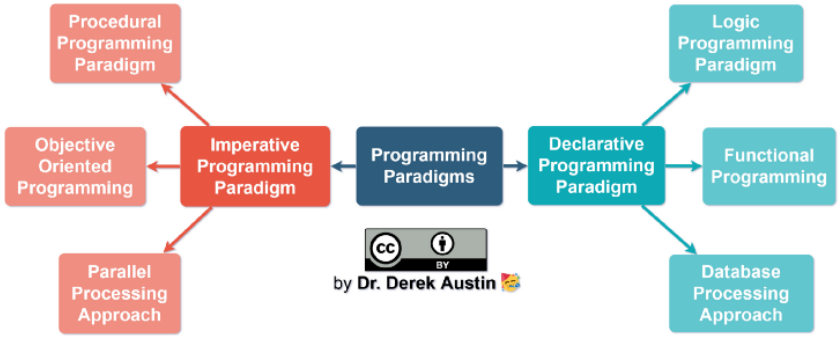


Figura 1. Schema dei paradigmi di programmazione più comuni.

grammatore specifica cosa deve essere fatto, piuttosto che come farlo. La programmazione funzionale ha dimostrato la sua utilità nell'elaborazione di dati paralleli e nella gestione delle operazioni asincrone, ed è oggi una componente chiave nello sviluppo di applicazioni distribuite e scalabili.

Lo Sviluppo del Pensiero Computazionale Oltre ai paradigmi di programmazione, la storia della programmazione informatica è accompagnata dal continuo sviluppo del pensiero computazionale. Questo concetto si riferisce alla capacità di formulare e risolvere problemi in modo che possano essere eseguiti da una macchina. Nel corso del tempo, il pensiero computazionale è diventato un pilastro dell'istruzione STEM (Scienza, Tecnologia, Ingegneria e Matematica) ed è stato incorporato in programmi educativi di tutto il mondo.

Il pensiero computazionale implica la scomposizione dei problemi complessi in problemi più piccoli, la ricerca di modelli e astrazioni, la progettazione di algoritmi efficienti e la capacità di tradurre soluzioni in istruzioni comprensibili dalla macchina. Queste abilità non sono solo importanti per i programmatori, ma sono diventate essenziali in molte discipline, dalla biologia alla fisica, dalla finanza alla psicologia.

In conclusione, la storia della programmazione informatica è un viaggio attraverso l'evoluzione dei paradigmi di programmazione e lo sviluppo del pensiero computazionale. Dall'imperativo all'orientato agli oggetti e al funzionale, ogni paradigma ha contribuito a plasmare il modo in cui concepiamo e sviluppiamo il software. Nel contempo, il pensiero computazionale è diventato una competenza fondamentale per affrontare le sfide del mondo moderno. Questa storia è un richiamo alla continua innovazione e all'importanza dell'educazione informatica nella formazione delle menti del futuro.

1.2 Paradigmi di programmazione

All'interno del mondo dei vari linguaggi di programmazione, si possono distinguere diversi *paradigmi* (Figura 1), ossia "stili" di programmazione distin-

ti[1]. Nei linguaggi di programmazione moderni molteplici stili possono essere presenti all'interno dello stesso linguaggio.

Un **paradigma** è una metodica con la quale ci si approccia alla programmazione. Esistono principalmente due paradigmi di programmazione:

- programmazione *imperativa*: l'enfasi è posta nello specificare le istruzioni che modificano lo stato del programma, al fine di risolvere uno specifico problema. Tra i linguaggi imperativi si distinguono quelli di tipo procedurale, ad esempio `C`, in cui l'idea di base è quella di fornire al calcolatore una sequenza di comandi, e quelli ti tipo object-oriented, ad esempio `Java, C++`, in cui un programma viene visto come un insieme di oggetti che interagiscono tra loro;
- programmazione *dichiarativa*: l'enfasi è posta nell'esprimere la logica di un calcolo, senza descriverne dettagliatamente il flusso di controllo. Tra i linguaggi dichiarativi si distinguono quelli ti tipo logico, in cui la logica del primo ordine è utilizzata sia per la rappresentazione che per l'elaborazione delle informazione, e quelli di tipo funzionale, in cui il flusso di esecuzione del programma consiste in una serie di valutazioni di funzioni matematiche.

Esempio 1.1. Semplice esempio di programma in paradigma imperativo per l'utilizzo di un ascensore.

1. attendere che l'ascensore sia arrivato al piano
2. aprire la porta dell'ascensore;
3. entrare nell'ascensore;
4. chiudere la porta;
5. spingere il tasto corrispondente al piano che si vuole raggiungere.

Equivalente programma in paradigma dichiarativo

1. se l'ascensore è arrivato e la porta è aperta, allora si può entrare;
2. se si vuole entrare nell'ascensore, bisogna aspettare che arrivi;
3. se si è entrati e la porta è aperta, allora si può chiudere;
4. se si è entrati e la porta è chiusa, allora si deve spingere il tasto corrispondente al piano che si vuole raggiungere.

[1] Un paradigma definisce un insieme di strumenti concettuali forniti da un linguaggio di programmazione per la stesura del codice sorgente di un programma, definendo dunque il modo in cui il programmatore concepisce e percepisce il programma

2 Nozioni fondamentali

2.1 Programma e computazione

Iniziamo dando la definizione di *programma*.

Definizione 2.1 (Programma)**.** Un **programma** è una descrizione eseguibile da un calcolatore di un metodo (ovvero un algoritmo) per il calcolo di un risultato (output) a partire da dati iniziali (input).

Intuitivamente, possiamo immaginare i nostri programmi come se fossero delle vere e proprie funzioni matematiche.

Nota bene (Connessione intuitiva tra programmazione e matematica)**.** Se si considera una funzione matematica f definita come come

$$f(x) = y$$

allora, usando il linguaggio della programmazione

- f è l'algoritmo;
- x è l'input;
- y è l'output.

Per scrivere i nostri programmi avremmo bisogno di utilizzare delle variabili di supporto (in numero arbitrario). Utilizzeremo queste variabili per scrivere delle *espressioni*, che definiremo più rigorosamente in seguito.

Definizione 2.2 (Variabile)**.** Una **variabile** è un *nome* associato ad un *valore*, modificabile.

Ad esempio nell'espressione $x = 17$ x è il nome della variabile e 17 il suo valore.

Nel contesto della programmazione imperativa, ogni nostro programma in esecuzione sarà sempre associato al suo "stato" . Lo stato ci permette quindi di tenere traccia del nostro programma, e del calcolo che sta effettuando. Noi lo definiamo naturalmente in base alle variabili che definiscono il programma.

Definizione 2.3 (Stato)**.** Lo **stato** è un insieme di variabili che che rappresentano quantità d'interesse per il programma.

Se un programma usa le variabili $x = 17$ ed $y = 12$, allora x ed y con i relativi valori costituiscono il suo stato.

L'esecuzione di un programma comporta dunque modificazioni successive al suo stato interno, in modo che il calcolo richiesto venga eseguito secondo quelle che sono le *istruzioni* del programma.

Lo stato iniziale del programma è quindi dato almeno dal suo *input* ed eventualmente da altre variabili. Lo stato finale è costituito dallo stato del programma alla fine dell'esecuzione, e da esse (o parte di esso) viene determinato l'output del programma. Il linguaggio di programmazione descrive, tramite quelle che possiamo chiamare *istruzioni* o *comandi*, i vari passaggi di stato (Figura 2).

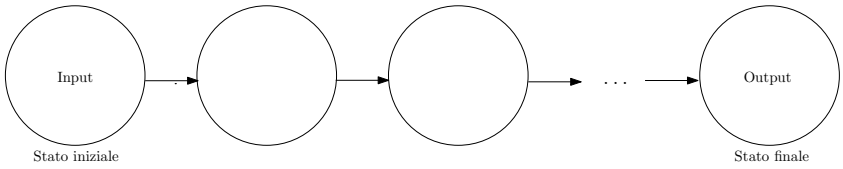


Figura 2. Un programma parte da un input per calcolare un output. Il suo stato, qui rappresentato come un cerchio, subisce successive modificazioni che possiamo immaginare essere l'effetto delle istruzioni stesse del programma.

Esempio 2.1. Si consideri un programma che, dati x e y in input, calcoli

$$f(x, y) = \log(x) + y^2 \, .$$

Supponiamo di "spezzare" il calcolo in vari passaggi, prima calcolando $\log(x)$, poi y^2, ed infine sommando i due valori. Una possibile modificazione dello stato interno durante l'esecuzione di questo programma è mostrata in Figura 3.

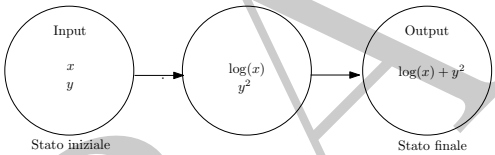


Figura 3. Stato interno durante l'esecuzione del programma dell'Esempio 2.1. Dopo l'input (x ed y), si calcolano $\log(x)$ a partire da x ed y^2 a partire da y (in un unico cambio di stato). Nello stato finale si sommano i due valori ottenendo $f(x, y)$ (output).

.

In generale, i programmi effettuano molteplici cambi di stato, sopratutto per calcoli che sono maggiormente complicati.

Qui, intuitivamente, denotiamo la complessità di un programma in base al numero di operazioni che il programma stesso deve eseguire. Nei corsi di algoritmica vedrete come questa intuizione possa essere formalizzata e contestualizzata per definire una teoria rigorosa della complessità algoritmica[2].

Esempio 2.2. Si consideri un programma che deve calcolare la somma di tre numeri. Assumiamo che l'input sia composto da tre variabili $x = 10, y = 5, z = 2$, da cui si debba calcolare

$$x + y + z = 17.$$

Una possibile esecuzione (Figura 4) di questo programma utilizza due

[2]Il lettore incuriosito potrebbe approfondire tramite la pagina Wikipedia sulla "Teoria della complessità computazionale".

variabili cosiddette di *appoggio* per memorizzare il parziale del calcolo

$$s = x + y = 10 + 5 = 15$$
$$ss = s + z = 15 + 2 = 17.$$

Notate che il nome di una variabile (ss) in questo caso non sia una singola lettera. In generale potremo dare nomi arbitrari alle variabili, e.g., "somma_parziale", purché costituiti da una singola parola.

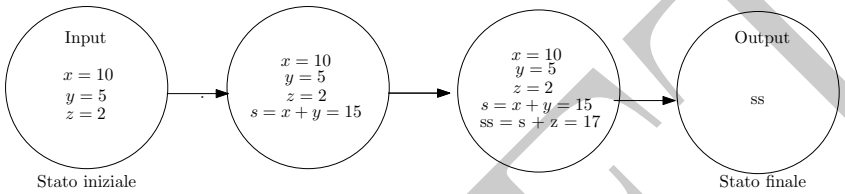


Figura 4. Stato interno durante l'esecuzione del programma dell'Esempio 2.2. Notare che certe variabili, ad esempio quelle di appoggio, compaiono solo in determinati punti dell'esecuzione del programma, mentre altre perdurano durante tutta l'esecuzione.

2.2 Assegnamento ed espressioni

Il **comando di assegnamento** è alla base della programmazione imperativa: l'assegnamento consente di modificare il valore di una variabile.

Definizione 2.4 (Assegnamento)**.** La sintassi del comando di assegnamento nel linguaggio di programmazione C è la seguente:

```
nome_variabile = expr;
```

In questa definizione:

- si nota l'utilizzo del "`;`", obbligatorio per delimitare la fine dell'assegnamento;
- `nome_variabile` è il nome delle variabile che vogliamo modificare;
- `expr` è una espressione che viene "valutata" per determinare il valore da assegnare alla variabile `nome_variabile`.

In un programma scrivere $x = 17$ comporta assegnare il valore 17 alla variabile x.

I programmi in `C` sono prevalentemente composti da **espressioni**, ovvero combinazioni di operatori e relativi operandi.

Possiamo dare una definizione di espressione più o meno rigorosa.

Definizione 2.5 (Espressione)**.** Le espressioni sono tutte quelle combinazioni di simboli che mettono assieme:

- nomi di variabili o costanti numeriche;
- operatori aritmetici o logici;
- nomi di funzioni.

Per il momento non sappiamo ancora cosa siano le funzioni, e gli operatori; nei prossimi capitoli si chiariranno queste definizioni.

Limitiamoci dunque a considerare le espressioni aritmetiche che utilizzano gli operatori di somma (+), sottrazione (−), moltiplicazione (⋆), divisione (/), modulo[3] (%), e le parentesi tonde. Espressioni quindi valide sono le seguenti (una per riga):

```
13 + 4
(17 - x) * (-1)
x
18%2
```

Disambiguare le espressioni. Se nell'espressione compaiono più operatori in quale ordine vengono interpretati?

Prendiamo ad esempio l'espressione $3-x+y$; chiaramente si potrebbe pensare ad entrambe queste possibilità

$$(3-x)+y \quad \text{v.s.} \quad 3-(x+y)\,.$$

Ad esempio per $x=3$ ed $y=7$, otteniamo due risultati diversi

$$(3-x)+y=7 \quad \text{v.s.} \quad 3-(x+y)=-7\,.$$

In realtà gli operatori sono soggetti a delle precise regole di precedenza, ovvero richiedono una convenzione per essere interpretate. In generale, a meno che non sia specificato diversamente tramite l'uso di parentesi, gli operatori $\star, /, \%$ hanno la precedenza su $+, -$.

Ciò che internamente viene utilizzato dal computer per svolgere i calcoli è un *albero di sintassi astratta* come quello mostrato in Figura 5. Dato l'albero di computazione, per stabilire l'ordine in cui vengono svolti i calcoli, a partire dalle foglie si risalgono i nodi interni fino ad arrivare alla radice, svolgendo le operazioni nell'ordine in cui si incontrano nel percorso.

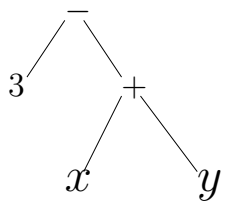


Figura 5. Albero di computazione di $3-x+y$ che mostra come la espressione sia equivalente a $3-(x+y)$; quindi prima viene valutato $x+y$, e successivamente il risultato viene sottratto da 3.

Esempio 2.3 (Una semplice computazione)**.** Si consideri il seguente problema: *con* 30000 € *si vogliono coprire le spese di acquisto di un'auto e il carburante necessario per un anno.*

Dopo aver scritto un programma per risolvere il quesito, vogliamo rispondere alle seguenti domande:

- quali sono le variabili di stato?
- qual è una possibile esecuzione di questo programma?

[3] Il modulo - normalmente in algebra indicato come mod - calcola il resto della divisone per un numero. Ad esempio, 18 % 2 fa 0, mentre 5 % 2 fa 1.

Per risponder a queste domande facciamo delle ipotesi. Si supponga che il costo dell'auto sia $c = 20000$ €, il costo della benzina per chilometro sia $b = 0.2$ €/km ed i chilometri percorsi in un anno siano $k = 10000$ km/anno.

Una possibile soluzione può essere quella di considerare come variabili di stato, oltre alle variabili di input c, b e k, una variabile tot in cui memorizziamo il totale calcolato. Spezzando il calcolo in due passaggi potremmo prima calcolare

$$tot = b \times k = 0.2 \times 10000 = 2000€/anno$$

per ottenere la spesa in benzina di un anno. Al passaggio successivo nell'esecuzione possiamo aggiungere il costo dell'auto, ovvero:

$$tot = tot + c = 2000 + 20000 = 2200€$$

La variabile tot definisce quindi l'output. Il nostro programma tutto assieme diventa:

$$tot = b \times k$$
$$tot = tot + c$$

e la sua esecuzione è mostrata in Figura 6.

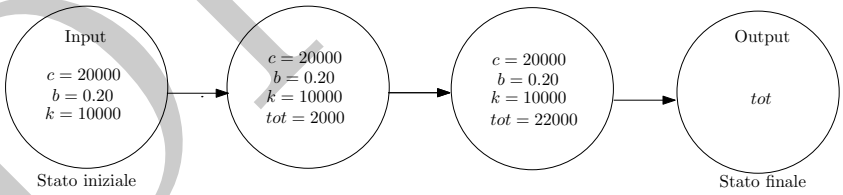


Figura 6. Stato interno durante l'esecuzione del programma dell'Esempio 2.3

Esempio 2.4 (Distanza tra due numeri)**.** Vogliamo scrivere un programma che calcoli la distanza tra due valori scalari a e b. In altre parole, vogliamo scrivere un programma che dati in input a e b calcoli $f(a, b) = |a - b|$.

Una possibile soluzione è cominciare assegnando ad una variabile d la differenza tra a e b, cioè:

$$d = a - b$$

e osservare che è uguale alla distanza tra a e b, cioè $d = a - b = |a - b|$, se $a \geq b$. Ovvero, se $d = a - b \geq 0$ allora $d = |a - b|$.

Tuttavia, se $d < 0$ significa che $b < a$. Quindi $d = a - b = -|a - b|$. Perciò, in questo secondo caso, se vogliamo che la variabile d alla

fine del programma contenga il valore corretto dobbiamo effettuare l'assegnazione:

$$d = (-1) \times d$$

abbiamo quindi le seguenti istruzioni che implementano il programma:

1. calcoliamo $a - b$ e lo assegniamo a d, cioè $d = a - b$
2. se $d \geq 0$ allora il programma termina;
3. se $d < 0$ allora assegniamo $d = (-1) \times d$ e poi il programma termina.

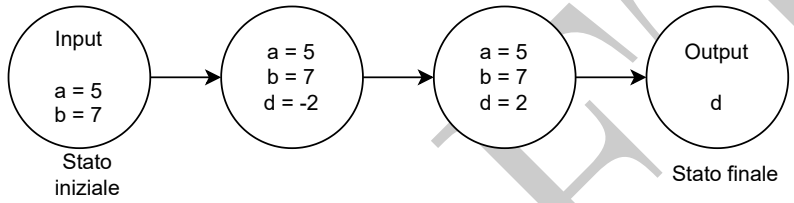


Figura 7. Stato interno durante l'esecuzione della prima soluzione nell'esempio 2.4 nel caso in cui $d < 0$.

Possiamo anche trovare una seconda soluzione con delle istruzioni diverse ma che arriva allo stesso risultato. Ovvero:

1. calcoliamo $a - b$
2. se $a - b \geq 0$ allora assegniamo $d = a - b$ ed il programma termina;
3. se $d < 0$ allora assegniamo $d = b - a$ e poi il programma termina.

Notiamo che se la condizione $a \geq b$ è necessaria per gestire il caso in cui $a = b$. Nel caso noi mettessimo solo la condizione $a < b$ allora il programma non sarebbe in grado di gestire il caso $a = b$.

Esempio 2.5 (Algoritmo di moltiplicazione del contadino russo)**.** Vediamo ora una tecnica per effettuare la moltiplicazione tra due numeri. Vogliamo, ad esempio, effettuare la moltiplicazione 146×37. Per farlo utilizzando questo particolare metodo, diviamo il numero 146 per 2 fino ad ottenere 1, arrotonando per difetto quando incontriamo un numero dispari. Allo stesso tempo, moltiplichiamo 37 per 2 lo stesso numero di volte. Scriviamo i risultati di ogni operazione nella seguente tabella.

146	37
73	74
36	148
18	296
9	592
4	1184
2	2368
1	4736

Da questa tabella eliminiamo tutte le righe di cui il valore che compare nella colonna sinistra è un numero pari.

73	74
9	592
1	4736

Se ora sommiamo tra loro i valori rimasti nella colonna di destra abbiamo:

$$74 + 592 + 4736 = 5402$$

che si può facilmente verificare sia proprio il valore della moltiplicazione 146×37. Si può anche verificare, con più fatica, che questa algoritmo è corretto in generale: omettiamo qui la dimostrazione per brevità.

Tramutiamo ora questo algoritmo in un programma. Vogliamo scrivere un programma che, dati in input due scalari $m > 0$ e $n > 0$, calcoli:

$$f(m, n) = m \times n$$

utilizzando il metodo del contadino russo. Avremo le seguenti istruzioni:

1. creiamo una variabile p a cui assegniamo 0, cioè $p = 0$, questa variabile conterrà nello stato finale il risultato della moltiplicazione;
2. se m è pari allora andiamo alla quinta istruzione;
3. assegniamo $p = p + n$;
4. se $m = 1$ allora il programma termina;
5. assegniamo $m = m/2$;
6. assegniamo $n = n \times 2$;
7. andiamo alla seconda istruzione.

Verifichiamo che queste istruzioni portino al risultato corretto con $m = 7$ e $n = 4$.

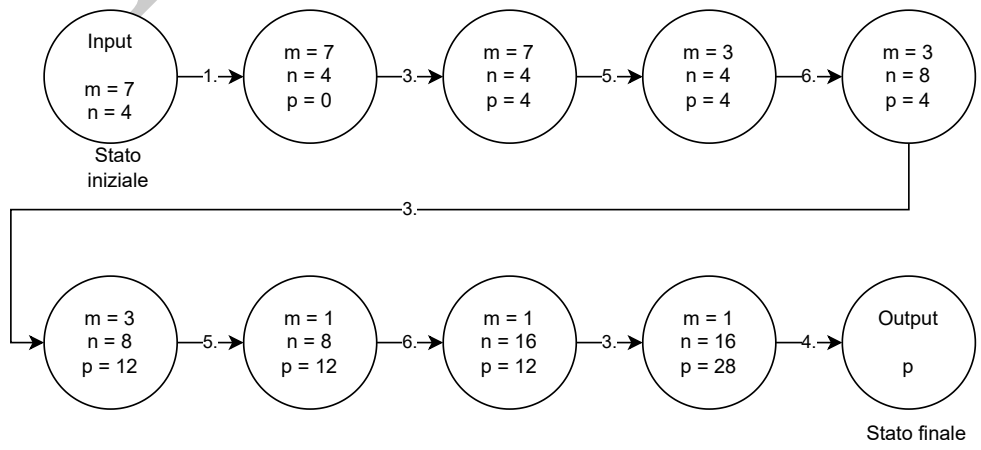


Figura 8. Stato interno durante l'esecuzione del programma nell'esempio 2.5 con $m = 7$ e $n = 4$. Nota che le istruzioni condizionali di cui la condizione non è soddisfatta e l'istruzione 7. sono omesse nel disegno per compatezza.

Nota bene. Provando a ripetere l'esempio del contadino russo con $m = 147$ e $n = 36$, si osserva che il numero di volte in cui dobbiamo ripetere le stesse istruzioni durante l'esecuzione aumenta rispetto al caso in cui $m = 7$ e $n = 4$.

Questo ci suggerisce che la duranta dell'esecuzione di un programma (che intuitivamente è proporzionale al numero di stati in cui esso si trova durante l'esecuzione) non dipende solo dal numero di istruzioni del programma, ma anche dagli input che diamo ad esso.

Quindi benchè lunghezza del programma sia fissata, quella della computazione è variabile e dipende dai valori di input n ed m.

2.3 Ambiente e memoria

Un computer ha una memoria fisica (i.e., hardware) chiamata Random Access Memory (RAM) il cui funzionamento verrà approfondito nel modulo B del corso di Programmazione.

Per **memoria**, in questa dispensa, intendiamo invece una memoria *logica*, coincidente con la memoria virtualmente accessibile sul computer. Stiamo quindi considerando un concetto "astratto" di memoria, senza prendere in considerazione le dinamiche legate al sistema operativo e l'hardware in questione. Questo modello della memoria non é tanto impreciso. Infatti, in un sistema operativo moderno, in ogni istante centinaia di programmi indipendenti sono in esecuzione concorrente, con ogni programma che ha a disposizione "virtualmente" tutta la memoria RAM a disposizione del computer.

In particolare la memoria è un oggetto indicizzabile tramite degli indirizzi - solitamente rappresentati in forma esadecimale; la codifica dei numeri verrà affrontata nel modulo B di questo corso. Lo stato di un programma viene quindi rappresentato come una coppia **ambiente-memoria**.

Figura 9. Rappresentazione schematica della memoria.

Definizione 2.6 (Stato come ambiente e memoria)**.** Definiamo lo stato di un programma come un ambiente ed una memoria, due strutture separate ma concettualmente collegate, tali che:

- l'ambiente, ρ, associa ad ogni variabile un indirizzo della memoria;
- la memoria, σ, associa ad ogni indirizzo un valore.

Nel nostro corso, indicheremo sempre gli indirizzi della memoria come l_0, l_1, l_2, ..., dove $l_i \in \mathcal{L}$.

Esempio 2.6. Si considerino due variabili $x = 10$ e $y = 20$, ovvero un programma contente assegnamenti tipo:

```
x = 10;
y = 20;
```

Lo stato del programma è rappresentato come in Figura 10.

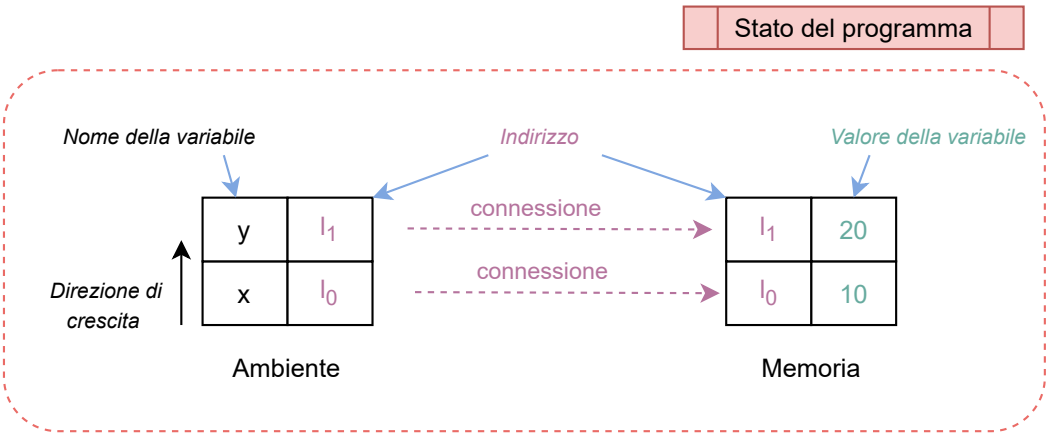


Figura 10. Stato del programma dell'Esempio 2.6, rappresentato come coppia ambiente (ρ) e memoria (σ). Ogni scatola rappresenta una associazione nell'ambiente per una variabile (x ed y), l_0 ed l_1 sono indirizzi e 20 e 10 i valori. Per accedere al valore della variabile x, in una qualunque espressione, faremmo $\sigma(\rho(x)) = 10$, passando quindi attraverso l'indirizzo $\rho(x) = l_0$.

Possiamo vedere ambiente e memoria come due funzioni. Se denotiamo con $\mathcal{V}$ lo spazio dei possibili valori memorizzabili in una memoria σ, e con $\mathcal{X}$ lo spazio dei nomi di variabili, allora avremmo

$$\rho : \mathcal{X} \mapsto \mathcal{L} \qquad \sigma : \mathcal{L} \mapsto \mathcal{V}\,.$$

Quindi, per conoscere il valore di una variabile $x \in \mathcal{X}$, bisogna calcolare la composizione $\sigma(\rho(x))$.

2.4 Dichiarare una variabile

Come già menzionato, una variabile è uno spazio di memoria che contiene un valore ed ha un nome/indirizzo associato.

Per poter essere utilizzata, una variabile va prima dichiarata; inoltre ogni variabile deve essere dichiarata una ed una sola volta.

Definizione 2.7 (Dichiarazione di variabile)**.** Esistono due modi per dichiarare una variabile, *semplice* o *con inizializzazione*, a seconda del fatto che essa sia anche inizializzata ad un valore specifico, oppure no. In C la sintassi è la seguente:

```
tipo nome_variabile; // semplice
tipo nome_variabile = expr; // con inizializzazione
```

Si noti che:

La dichiarazione `int x;` corrisponde a $x \in \mathbb{I}$.

- in entrambi i casi la dichiarazione termina con "`;`" obbligatorio;
- `expr` è una espressione che viene valutata per determinare il valore da assegnare a `nome_variabile` (inizializzazione).

L'inizializzazione ci pone davanti ad un concetto molto importante, ossia quello del *tipo* di una variabile.

Il tipo definisce quello che la variabile può contenere - i.e., il suo dominio matematico di riferimento - e la quantità di memoria (in byte) che la variabile occupa[4]. In certi linguaggi, ad esempio in C, detti "fortemente tipati", ad ogni variabile corrisponde uno ed un solo tipo associato. In altri linguaggi, ad esempio in Python, questa caratteristica viene meno.

Ad ogni variabile in `C` deve quindi essere associato un tipo, dichiarato contestualmente alla dichiarazione della variabile. Per quanto riguarda il nostro semplice corso, possiamo restringerci a considerare i seguenti tipi:

- `int`: numero intero (positivo o negativo, quindi in $\mathbb{Z}$);
- `float, double`: numero in virgola mobile (un numero reale, in $\mathbb{R}$);
- `char`: carattere (ad esempio `'A'`, `'a'`, etc. oppure `'1'`, `'2'`, etc.).

Senza scendere in dettagli, `double` è il doppio più preciso di `float`[5]. Un aspetto importante della programmazione è quindi la scelta dei tipi giusti di variabili, a seconda di ciò che vogliamo modellare con il nostro programma.

Esempio 2.7. Considerando ancora il problema dell'Esempio 2.3, alcune di queste quantità, e.g., il costo della benzina per chilometro, dovranno essere necessariamente memorizzate con la virgola. Inoltre, il risultato del calcolo parziale (`tot`), essendo il risultato di moltiplicazioni tra numeri interi e con la virgola, dovrà necessariamente avere la virgola.

Mettendo insieme queste considerazioni possiamo scrivere questo semplice programma `C`.

```
/* dichiarazioni variabili */
int k = 10000;
float b = 0.2;
int c = 20000; // 3
float tot = 0; // 4

/* computazioni con le variabili */
tot = k*b; // 5
tot = tot + c; // 6
```

[4] Nel corso del modulo B vedrete aspetti specifici riguardanti la codifica binaria ed esadecimale di numeri interi e reali.

[5] Per i curiosi, `float` è un numero rappresentato su 32 bit a singola precisione, con 1 bit per il segno, 8 per l'esponente, e 23 per il valore, i.e. `float` ha quindi 7 cifre decimali per la precisione. `double` usa 64 bit in doppia precisione, con 1 bit per il segno, 11 per l'esponente, e 52 per il valore, i.e. `float` ha quindi 15 cifre decimali per la precisione.

Lo stato di questo programma è rappresentato in Figura 11. Si noti che `tot` viene dichiarata ed inizializzata a 0 avremmo potuto evitare di inizializzare questa variabile?

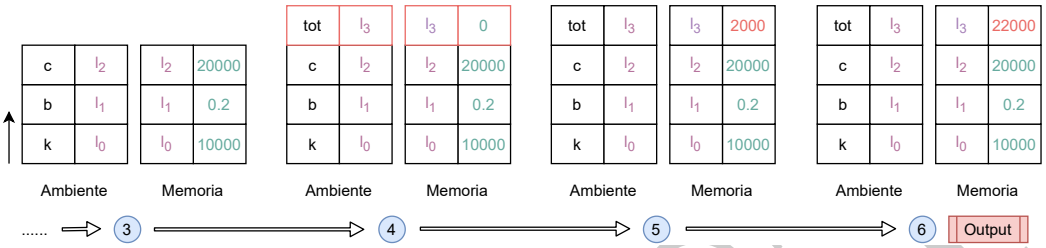
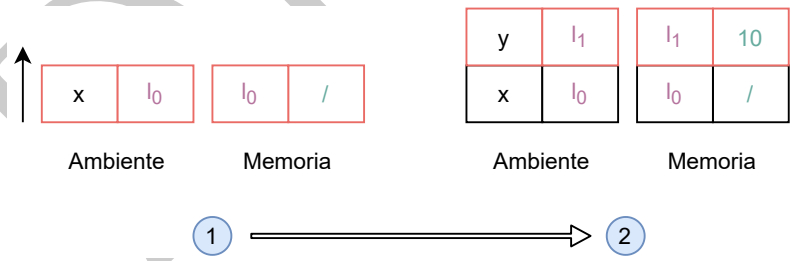


Figura 11. Stato del programma dell'Esempio 2.7, ai punti indicati dalle da 3 a 6. I cambi sullo stato del programma sono annotati in rosso. Al punto 4 una nuova variabile viene creata, il cui valore cambia ai punti 5 e 6. Si noti che ambiente e memoria crescono in verticale, cosi come gli indirizzi l_0, l_1, ...

Nota bene. Si consideri il seguente programma:

```
int x; // 1. dichiarazione semplice
int y = 10; // 2. dichiarazione con inizializzazione
y = y + x; //3. valutazione espressione
```

Lo stato del programma, ai punti 1 e 2, è:



Può essere valutata l'espressione al punto 3, $y + x$?

No. Essendo x indefinito - poiché dichiarato senza inizializzazione (rappresentato con il simbolo / nella memoria) - anche $y + x$ rimane una espressione indefinita, anche se y è definito. Quindi quella espressione non si può valutare con successo; un compilatore dovrebbe rendersene conto e dare un errore in fase di compilazione del programma. Bisogna sempre assicurarsi che alle variabili usate in un'espressione sia stato associato in precedenza un valore!

Quindi, più nel dettaglio, se consideriamo anche il caso in cui le variabili non siano inizializzate, la memoria è una funzione $\sigma : \mathcal{L} \to \mathcal{V} \cup \{\perp\}$, dove $\perp$ indica un valore speciale assegnato alle variabili non inizializzate.

Esercizio 2.1. Utilizzando la notazione vista negli esempi precedenti, disegnare ambiente e memoria del seguente frammento di codice `C`, a ciascuna delle istruzioni del programma.

```c
/* dichiarazioni semplici */
int x;
int y;
int z, w; // vedi sotto

/* assegnamenti */
x = 1;
y = 3;
z = x + y;
w = 2 * z + (z - 1)
```

Si noti che la dichiarazione

```c
int z, w;
```

è sintatticamente equivalente alle due dichiarazioni

```c
int z;
int w;
```

3 Istruzioni condizionale ed iterative

Fino ad ora abbiamo immaginato il flusso di esecuzione del nostro programma come sequenziale, dall'alto verso il basso, senza ripetizioni. In realtà esistono comandi che permettono di saltare o ripetere un numero arbitrario di volte certe porzioni di un programma.

3.1 Istruzione condizionale `if`

Definizione 3.1. (If-then-else) L'istruzione condizionale `if` ha la seguente sintassi:

```
if(condizione)
{
    comando1;
}
else
{
    comando2;
}
```

dove `condizione` è un'*espressione logica* a cui si può associare valore "vero" o "falso", mentre `comando1` e `comando2` sono due programmi.

Nota bene. Per essere valutabile, `condizione` deve essere corretta, nel senso che tutte le variabili che vi compaiono devono essere correttamente inizializzate. Invece, "vero" o "falso" in `C` non esistono - mentre esistono i tipi Booleani in altri linguaggi. Per questo motivo, ci si riferisce sempre al fatto che vero sia in realtà 0, mentre falso sia un valore maggiore di 0.

Una volta valutata la condizione dell'`if`, a seconda che questa sia vera o falsa il programma continua differentemente con `comando1` oppure con `comando2` - da cui ciascun comando può essere una serie complessa di istruzioni, i.e., un altro programma vero e proprio, ad esempio un altro `if`! Quindi rami diversi del programma vengono eseguiti, come mostrato in Figura 12.

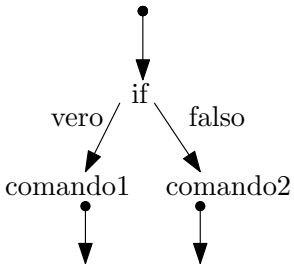


Figura 12. Funzionamento del comando condizionale `if`.

In quest'ottica, il flusso del programma non è un'esecuzione lineare, ma ci sono separazioni (i.e., branches) computazionali in corrispondenza delle strutture condizionali. Si noti che si possono anche inserire comandi `if` senza il corrispondente ramo `else`: il comando viene eseguito se e solo se la condizione è vera. In codice:

```
if(condizione)
{
    /* se la condizione è falsa, il comando non viene eseguito */
    comando;
}
```

Nota bene. Se il comando all'interno della struttura condizionale consiste di una singola istruzione si possono omettere le parentesi graffe.

Le strutture condizionali possono quindi essere annidate:

```
if(condizione1)
{
    /* qui vale condizione1 */
    if(condizione2)
    {
        /* qui valgono sia condizione1 che condizione2 */
        comando1;
    }
    else
    {
        /* qui vale condizione1 ma non condizione2 */
        comando2;
    }
}
else
{
    /* qui non vale condizione1 */
    comando3;
}
```

Esempio 3.1. Consideriamo il seguente codice, che calcola il massimo tra i valori di due variabili:

```
int x = 10;
int y = 10;
int max; //var. dove verrà memorizzato il massimo

/* condizionale */
if(x>y)
    max = x; //comando1
else
    max = y; //comando2
```

Lo stato interno di questo programma è rappresentato in Figura 13.

> **Nota bene.** In questo caso viene eseguito `comando2`, poiché non è vero che $x > y$ (i.e., `x>y` è falsa), in quanto
>
> $$\sigma(\rho(x)) \not> \sigma(\rho(y))$$
>
> dato che $10 \not> 10$. La negazione di $>$ non è $<$ bensì $\leq$!

max	l_2
y	l_1
x	l_0

l_2	~~?~~10
l_1	10
l_0	10

Figura 13. Stato del programma dell'Esempio 3.1

Esercizio 3.1. Dato il seguente frammento di codice, disegnarne la memoria e stabilire quale valore assume `max`:

```
int x = 33;
int y = 10;
int max; //var. dove verrà memorizzato il massimo

/* condizionale */
if(x>y)
```

```
    max = x; //comando1
else
    max = y; //comando2
/* quanto vale max? */
```

3.2 Espressioni logiche

Abbiamo precedentemente parlato di espressioni mediante operatori quali somma, prodotto etc. Adesso introduciamo le *espressioni logiche* che, ovviamente, si utilizzano per determinare i flussi di esecuzioni di comandi più complessi quali gli `if`.

In generale le espressioni logiche hanno la sintassi:

```
espressione1 operatore espressione2
```

laddove `operatore` è un operatore tra:

- *disuguaglianze*: `<`, `>`, `<=`, `>=` (dove `<=` e `>=` stanno per $\leq$ e $\geq$, rispettivamente);
- *uguaglianza*: `==`;
- *diversità*: `!=` (che sta per $\neq$)

Nota bene. Attenzione a non confondere l'operatore di uguaglianza `==` con l'operatore di assegnamento `=`. Tecnicamente l'assegnamento è un comando come gli altri, e quindi ha un valore di ritorno[a] opportuno. Per questo motivo il singolo `=` può essere utilizzato in una condizione logica, ma non avrebbe ovviamente il significato inteso.

[a] Questa terminologia diventerà più chiara quando introdurremo le funzioni.

Esempio 3.2. Alcuni semplici esempi di espressioni logiche sono:

```
x == 0
(x + 3) != (y - 2)
x >= y/7
x < 4
```

Si noti che `x >= y/7` equivale a `x >= (y/7)`, in termini di precedenza degli operatori.

Le espressioni logiche possono inoltre contenere connettivi logici, altrimenti chiamati operatori dell'algebra booleana:

- *congiunzione* `&&` ("and" logico, $\wedge$ in algebra);
- *disgiunzione* `||` ("or" logico, $\vee$ in algebra);
- *negazione* `!` ("not" logico, $\neg$ in algebra);

Il loro significato è intuitivo: le congiunzioni sono vere se entrambi gli operandi sono veri contemporaneamente, le disgiunzioni sono vere quando anche uno solo degli operandi è vero. Infine la negazione é vera quando l'operando è falso.

A questi connettivi possiamo quindi associare delle *tabelle di verità* dove 0 corrisponde a vero ed 1 a falso, sono le seguenti.

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

La tabella riporta in ogni colonna il valore di verità della formula (o predicato) composta con il connettivo, e possiamo quindi utilizzare i connettivi logici per creare formule di arbitraria complessità.

Esempio 3.3. Semplice espressione logica che valuta vero se e solo se una variabile x sta nell'intervallo $[-4, 5]$ (ovvero $-4 \leq x \leq 5$):

```
(x >= -4) && (x <= 5)
```

Oppure, espressione logica che valuta vero se è solo una variabile x sta fuori dal medesimo intervallo:

```
(x < -4) || (x > 5)
```

Discutere del valore di verità di: `(x < -4) && !(x <= 5)`.

Vediamo ora un uso pratico di condizioni logiche composte, e come possiamo scrivere il codice di un programma in modo diversi, pur calcolando lo stesso valore. L'obiettivo è scrivere un programma che assegni 1 ad una variabile z se e solo se due altre variabili - x ed y - sono concordi in segno, -1 se sono discordi.

```
// due variabili di esempio, con i valori opportuni
int x = 10;
int y = -2;

/* var. per memorizzare il risultato */
int z; //1

/* condizionale */
if(
    ((x > 0) && (y > 0)) ||
    ((x < 0) && (y < 0))
    )
    /* x, y concordi */
    z = 1;
else
```

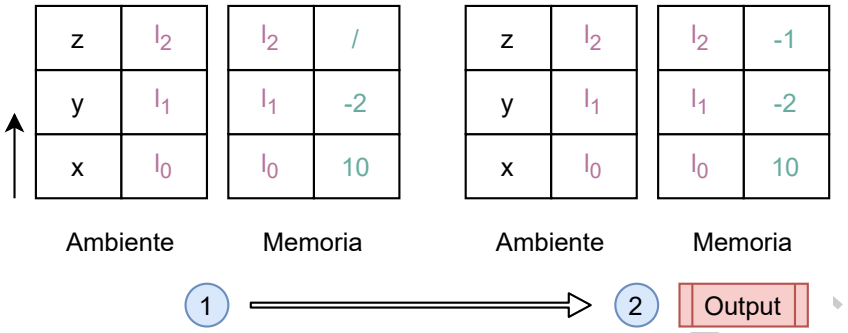


Figura 14. Stato del programma che assegna ad una variabile `z` il valore 1 se e solo due variabili `x` ed `y` sono concordi in segno. Al punto 1 la variabile `z` è stata dichiarata ma non ancora inizializzata, al punto 2 a questa viene assegnato un valore.

```
    /* x, y discordi */
    z = -1; //2
```

Lo stato interno di questo programma è rappresentato in Figura 14.

A questo punto, tentiamo di scrivere lo stesso programma usando un solo ramo `if`. Quello che faremo è una prassi comune per evitare di utilizzare entrambi i rami `if-else` nel caso in cui il programma sia relativamente semplice. Assumiamo quindi di avere una risposta a priori al nostro calcolo (e.g., stabilire che x ed y sono concordi, per cui $z = 1$, oppure viceversa), quindi assegnare un valore a z ed agire di conseguenza cambiandolo in caso di evidenza contraria.

```
int x = 10;
int y = -2;

/* suppongo che x, y siano discordi */
int z = -1;

/* condizionale */
if(
    ((x > 0) && (y > 0)) ||
    ((x < 0) && (y < 0))
    )
    /* se e solo se x, y concordi */
    z = 1; //modifico z
```

Nota bene (Chiarezza del codice)**.** Lo stesso programma può essere scritto utilizzando una variabile d'appoggio rendendo il codice più comprensibile, scelta utile soprattutto se le condizioni diventassero

complesse.

```
int x = 10;
int y = -2;

/* suppongo che x, y siano discordi */
int z = -1;

/* valori concordi vs discordi */
int concordi_positivi = ((x > 0) && (y > 0));
int concordi_negativi = ((x < 0) && (y < 0));

/* condizionale */
if(concordi_positivi || concordi_negativi )
    /* se e solo se x, y concordi */
    z = 1; //modifico z
```

Un modo ancora più elegante e succinto di ottenere lo stesso obiettivo richiede di sfruttare una proprietà del segno del prodotto $x \cdot y$. Ovvero il fatto che il prodotto di due numeri concordi in segno è sempre positivo, viceversa il prodotto di due numeri discordi è sempre negativo ovvero:

$$x > 0, y > 0 \implies x \cdot y > 0$$
$$x < 0, y < 0 \implies x \cdot y > 0$$

Le proposizioni logiche qui sopra si leggono come implicazioni logiche (simbolo $\implies$): "se x è positiva ed y è positiva, allora il prodotto $x \cdot y$ è positivo", ed idem per il caso opposto.

Sfruttando questa semplice intuizione, in codice:

```
int x = 10;
int y = -2;

//memorizzo il prodotto di x ed y
int w = x*y;

int z;

/* condizionale */
if(w>0) //x ed y sono concordi
    z = 1;
else
    z = -1
```

Si noti che anche in questo caso potrei assumere una risposta a priori per il confronto, e partire per esempio con $z = 1$ o $z = -1$. Lo stato interno del

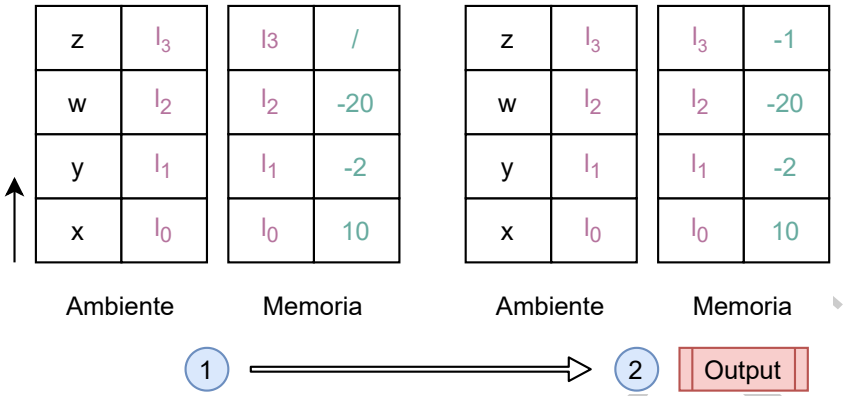


Figura 15. Stato del programma durante l'esecuzione dell'ultima soluzione proposta per il calcolo della concordanza del segno mediante il prodotto di due valori scalari. In questo caso al punto 1 la variabile `w` viene inizializzata con $w = x \cdot y = -20$, mentre `z` rimane non inizializzata; al punto 2 invece `z` assume valore -2.

programma durante l'esecuzione di questo codice è rappresentato in Figura 15.

Esercizio 3.2. Vogliamo creare un connettivo logico di nome *xor* ('or esclusivo', spesso indicato come $x \oplus y$) che sia vero solamente se al più uno dei suoi due operandi è vero. La tabella di verità di questo connettivo è la seguente.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Trovare un modo per esprimere $x \oplus y$ in funzione dei tre operatori $\wedge$, $\vee$ e $\neg$.

3.3 Istruzione iterativa `while`

Ci sono situazioni in cui si vuole ripetere lo stesso frammento di programma per molteplici volte, quindi consecutivamente. In questi casi dobbiamo utilizzare le **istruzioni iterative**.

Nel caso specifico che vedremo in questa sezione *non conosciamo a priori* il numero di volte in cui tale frammento verrà ripetuto.

Esempio 3.4 (Conto alla rovescia)**.** Cominciamo con il motivare la necessità di tali costrutti. Pensiamo di dover scrivere un programma che, dato un valore iniziale per una certa variabile intera $x > 0$, faccia il *conto alla rovescia*, ovvero esegua l'istruzione `x = x-1` fino a che `x = 0`.
Supponiamo di risolverlo nel seguente modo, assumendo che x valga 6:

```
// Dichiaro x
```

```c
int x = 6;

x = x - 1; //x=5
x = x - 1; //x=4
x = x - 1; //x=3
x = x - 1; //x=2
x = x - 1; //x=1
x = x - 1; //x=0
```

E' chiaro che un programma come questo dipende dal valore che ha inizialmente la variabile `x`. Se `x` valesse 18, dovrei ripetere l'assegnamento `x = x-1` per 18 volte.

Quindi questo programma non è generalizzabile, perché non funziona per ogni possibile valore di x. Quindi, se quello che vogliamo è un meccanismo che funzioni per ogni valore di `x`, dobbiamo utilizzare una altra tipologia di costrutto.

Quando ci troviamo in casi come quello dell'Esempio 3.4, ci occorrono i *comandi iterativi*, di cui l'istruzione `while` fa parte.

Definizione 3.2 (While)**.** L'istruzione iterativa `while` ha la seguente sintassi:

```c
/* fino a che vale condizione */
while(condizione)
{
    /* esegui comando */
    comando;
}
```

A parole, il comando iterativo `while` esegue il programma `comando` fin tanto che `condizione`, che è un'espressione logica, è vera, come schematicamente mostrato in Figura 16. Qui per `comando` intendiamo una sequenza qualunque di espressioni e altri costrutti C, come quelli visti fino ad ora.

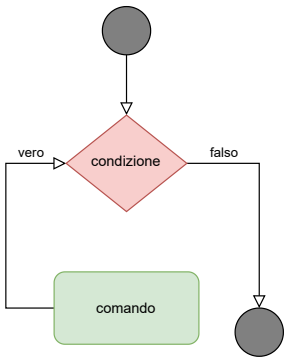


Figura 16. Diagramma di flusso dell'istruzione iterativa `while`.

Esempio 3.5 (Conto alla rovescia con while)**.** Utilizzando l'istruzione iterativa `while`, il programma dell'Esempio 3.4 si può risolvere nel seguente modo:

```c
int x = 6;
/* iterativo */
while(x>0)
{
    /* ad ogni iterazione modifico il valore di x e controllo la condizione */
    x = x - 1;
}
```

Così facendo, ripetiamo l'operazione `x = x - 1` fino a che vale che `x >`

`0`. Essendo `x` un intero positivo, questa condizione risulterà falsa quando `x == 0`, che è esattamente il comportamento che vogliamo descrivere. Per sua natura, questa soluzione funziona ovviamente per qualunque valore sia assegnato ad `x`.

Un sacco di funzioni matematiche semplici possono essere esplicitamente definite mediante una iterazione. Vediamo una delle più semplici, il fattoriale.

Esempio 3.6 (Fattoriale)**.** Il fattoriale (simbolo !) di un numero naturale x è definito come:

$$1! = 1$$
$$n! = n \cdot (n-1)! \qquad \text{se } n > 1$$

Un programma per calcolare il fattoriale di un numero naturale n deve fare almeno (o esattamente?) n operazioni. Dobbiamo quindi definirlo con un comando almeno iterativo:

```
// elemento di cui vogliamo calcolare il fattoriale
int n = 5;

int f = 1; // valore di partenza del fattoriale

/* iterazione */
while(n>1)
{
    f = f * n;
    n = n - 1;
}
```

Ad ogni iterazione moltiplichiamo `f` per il valore corrente di `n` e decrementiamo di 1 la variabile `n`. Si noti che `x` è stata inizializzata ad 1 in quanto questo l'*elemento neutro* per il prodotto, ossia quell'elemento per cui vale la seguente proprietà

$$\forall x. 1 \cdot x = x$$

Si ragioni del perché proprio questo numero sia stato utilizzato per inizializzare `f`. Lo stato interno del programma durante l'esecuzione di questo codice è rappresentato in Figura 17.

Il concetto di iterazione con `while`, che dobbiamo ulteriormente chiarire, ci permette di vedere la scrittura del nostro primo vero algoritmo.

Esempio 3.7 (Calcolo MCD con il metodo di Euclide)**.** L'algoritmo di Euclide è un metodo per calcolare il *massimo comune divisore* (MCD) tra due interi `x` ed `y`.

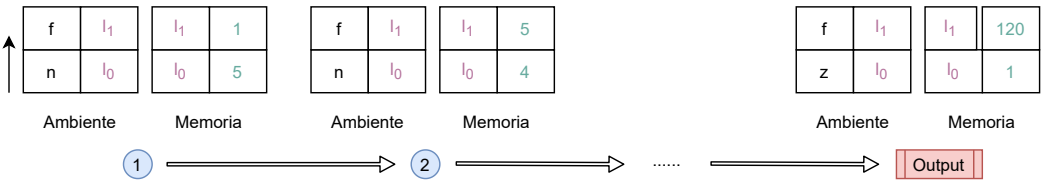


Figura 17. Rappresentazione dello stato interno del programma dell'Esempio 3.6. Ad ogni iterazione sia `f` che `n` vengono modificati. Il programma si arresta quando `n == 1`, che è il caso speciale della definizione del fattoriale.

A parole, la procedura è la seguente:

- parte con due numeri in input, appunto $x > 0$ ed $y > 0$, interi;
- sottrae il numero più piccolo tra `x` ed `y` dal più grande;
- continua, ossia ripete, la sottrazione del punto precedente fino a che i due numeri $x > 0$ ed $y > 0$ non sono uguali;
- il valore trovato - ovvero quello che rende `x` ed `y` uguali - è il MCD.

Gli strumenti che abbiamo sono sufficienti a scrivere un codice in `C` che implementi questo algoritmo:

```
int x = 30;
int y = 12;

int mcd; //variabile d'appoggio, in cui memorizzeremo il risultato

/* fintanto che i due numeri sono diversi */
while(x != y)
{
    /* sottrarre il numero più piccolo dal più grande */
    if(x>y)
        x = x - y;
    else
        y = y - x;
}

mcd = x; // a questo punto x ed y sono uguali
```

> **Nota bene.** Che ruolo hanno le assunzioni sulla positività di x ed y? Se non fossero garantite, il valore dei due numeri cresce o decresce? Si costruisca un esempio di computazione, omettendo per ora il disegno della memoria, visto che lo introdurremo più avanti. Dovrebbe essere possibile dimostrare che l'algoritmo di Euclide non termina (cioè non esce mai dal ciclo).

A questo punto possiamo chiarire meglio il senso del costrutto iterativo `while`. Dovrebbe infatti essere semplice notare che l'Esempio 3.7 sia un caso di **iterazione indeterminata**, ovvero un'iterazione della quale a priori non sappiamo

quante volte verrà eseguito il corpo del comando `while`.

La caratteristica fondamentale del costrutto `while` è quindi quella di essere utilizzato quando il numero di iterazioni del corpo del comando non può essere determinato.

Ci sono casi in cui, al contrario, il numero di iterazioni è conosciuto a priori: se vogliamo calcolare il fattoriale di un numero `n`, come nell'Esempio 3.6, sappiamo che il numero di iterazioni è esattamente `n`. In queste situazioni siamo davanti ad una **iterazione determinata**, per la quale abbiamo un altro tipo di comando iterativo.

3.4 Istruzione iterativa `for`

Diamo adesso la definizione di un'altra istruzione iterativa, il `for`, che è opportuno utilizzare in caso di iterazioni determinate.

Definizione 3.3 (Comando iterativo `for`)**.** Nel caso di iterazione determinata si usa il comando `for`, la cui sintassi è:

```
for(comando1; espressione; comando2)
{
    comando3;
}
```

Il suo funzionamento consiste nelle seguenti computazioni:

1. si esegue `comando1` (chiamato *inizializzazione*);
2. si valuta `espressione` (ovvero la condizione del `for`):
   - se `espressione` è falsa, l'esecuzione del `for` termina;
   - se `espressione` è vera: (*i*) si esegue `comando3`, il *corpo* del `for`, (*ii*) si esegue `comando2` (chiamato aggiornamento) e (*iii*) si salta al punto 2.

Una rappresentazione schematica della sintassi del comando iterativo `for` è mostrata in Figura 18.

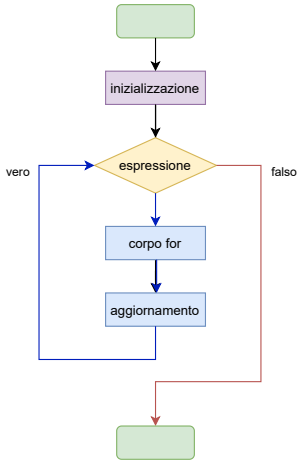


Figura 18. Flow chart del comando iterativo `for`.

Esempio 3.8 (Fattoriale usando il `for`)**.** Come già menzionato, nel caso di iterazioni determinate è opportuno utilizzare il `for` anziché il `while`. Il seguente frammento di codice mostra come calcolare il fattoriale di un intero `n` utilizzando il `for`.

```
int n = 5;
int f = 1;
```

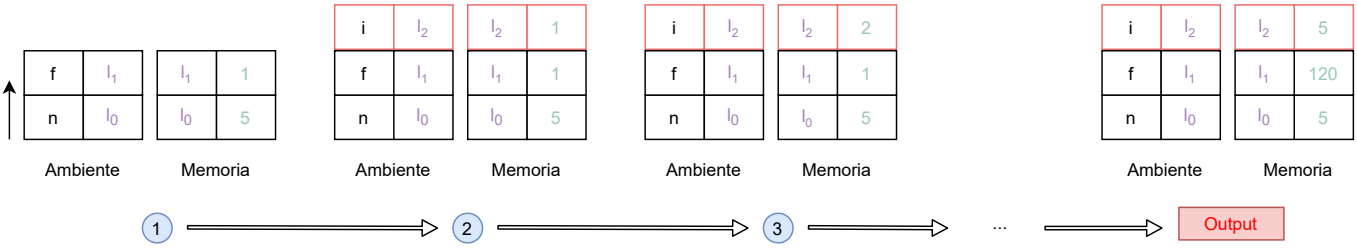


Figura 19. Stato del programma dell'Esempio 3.8. I cambi sullo stato del programma sono annotati in rosso.

```
/* iterazione con for */
for(int i=1; i<=n;  i=i+1)
{
    f = f*i;
}
```

Alcune considerazioni sulla logica dell'algoritmo sono analoghe rispetto a prima. Ad esempio, come già detto nell'Esempio 3.6, la variabile `f` viene inizializzata ad 1 essendo questo l'elemento neutro del prodotto.
Lo stato del programma durante l'esecuzione di questo codice è rappresentato in Figura 19.

Esercizio 3.3. Si consideri il contatore alla rovescia visto con il `while`. Che tipo di iterazione caratterizza quel calcolo? Si motivi e scriva l'implementazione corretta.

3.5 Blocchi di codice e pile di frame

Fino ad ora abbiamo utilizzato inconsciamente la nozione di *blocco*, ossia un pezzo di codice scritto tra parentesi.

```
{// blocco (apertura)

    // codice dentro al blocco

} // blocco (chiusura)
```

Definizione 3.4 (Blocco)**.** Un **blocco** in `C` è una porzione di codice racchiusa tra parentesi graffe. Il blocco consente di considerare una sequenza di comandi come se fosse un unico comando.

I blocchi possono anche essere annidati, vediamo ad esempio la dichiarazione di due variabile in due blocchi distinti (riprenderemo questo argomento a breve):

```
{// primo blocco
    int x = 10;

    {// secondo blocco
```

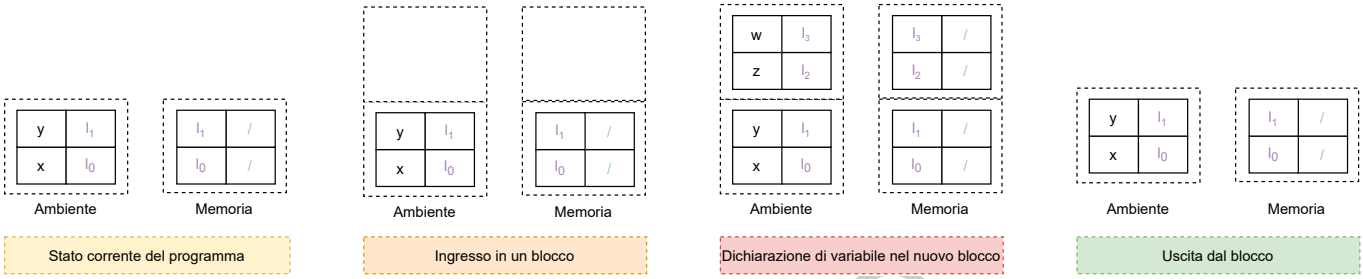


Figura 20. Pila di frame e sue modificazioni all'entrata e uscita da un blocco. Quando si entra in un nuovo blocco, un frame di ambiente e uno di memoria vengono creati sopra a quelli già esistenti; tutte le variabili che vengono dichiarate all'interno del nuovo blocco vivono nel nuovo frame; quando si esce da un blocco, sia il frame di ambiente che di memoria relativi a quel blocco vengono eliminati.

```
        int y = 10;

    } // fine secondo blocco
} // fine primo blocco
```

Come avrete notato i blocchi compaiono nella definizione sia dei comandi condizionali, che negli iterativi.

I blocchi ci servono per andare a raffinare i nostro modello della memoria/ambiente. Sfruttando la definizione di blocco appena fornita, possiamo descrivere lo stato interno di un programma come una sequenza di **frames** (singolare "frame"), ovvero di frammenti di memoria corrispondenti a blocchi di codice, che vengono continuamente creati o distrutti. I frames vengono costruiti "verticalmente", cioe' crescono in memoria verso l'alto, e decrescono verso il basso. Questo permette di realizzare una struttura che si chiama LIFO, ovvero *Last In First Out*, in cui il prossimo elemento che viene distrutto è l'ultimo che viene creato.

Il modello astratto di ambiente e memoria è quindi una pila (in inglese *stack*), in cui gli elementi (in questo caso i frames) che vengono eliminati per primi sono quelli che sono stati inseriti più tardi (intuitivamente, una pila cresce dal basso verso l'alto e decresce dall'alto verso il basso). L'operazione di inserire un elemento in una pila si chiama *push*, quello di estrarlo si chiama *pop* (TODO FIGURA). Il funzionamento dei frame è schematicamente illustrato in Figura 20.

Più precisamente, lo stato interno di un programma è quindi costituito da un ambiente che è una pila di frame di ambiente e da una memoria che é una pila di frame di memoria.

Esempio 3.9. Si consideri il seguente frammento di codice, contenente blocchi annidati:

```
/* primo blocco */
{
```

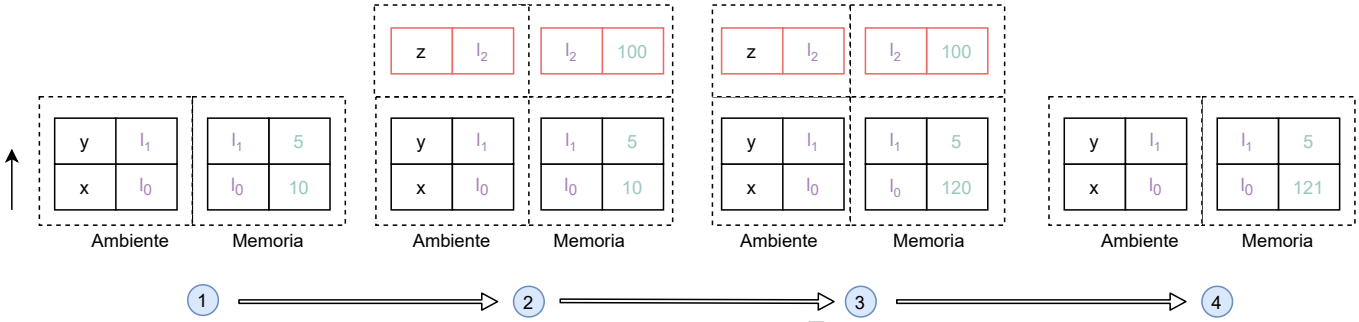


Figura 21. Stato interno del programma dell'Esempio 3.9. I blocchi tratteggiati rappresentano i frame di ambiente e memoria.

```
    int x = 10;
    int y = 5;

    x = x + y;
    /* secondo blocco (annidato) */
    {
        int z = 100;
        x = x + y + z;
    }
}
```

In Figura 21 è rappresentato lo stato interno del programma durante dell'esecuzione di questo codice.

I blocchi ci permettono di introdurre il concetto fondamentale di *variabili locali e globali*. Si consideri il seguente frammento di codice:

```
/* primo blocco */
{
    int x = 5;
    int y = 10;

    x = x + y; //1

    /* secondo blocco */
    {
        int x = 100; //variabile locale, 2
        x = x + y; //3
    }
    x = x + 1; //4
}
```

La variabile `x` dichiarata all'interno del secondo blocco, nonostante abbia lo stesso nome della variabile `x` dichiarata nella prima riga di codice è **locale** al blocco, ovvero viene eliminata una volta che si è fuori dal blocco in cui viene dichiarata. Da notare che all'interno del secondo blocco, le modifiche apportate alla variabile `x` hanno effetto sulla variabile locale `x`. La variabile `y` si chiama invece **globale**, quando ci si riferisce al secondo blocco (perché non viene definita in quel medesimo blocco).

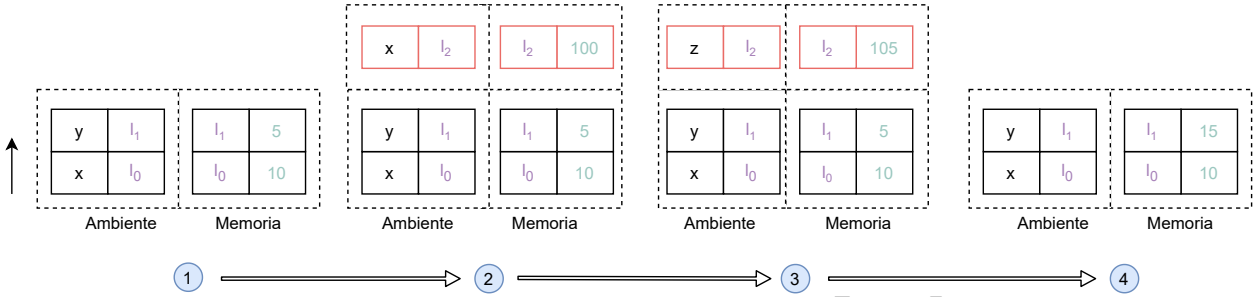


Figura 22. Stato di un programma in presenza di una variabile (`x`) locale ad un blocco. Al punto 2 viene dichiarata la variabile locale `x` in un nuovo frame, al punto 3 viene modificata come prescritto dal codice, infine al punto 4, quando ormai siamo fuori dal secondo blocco, viene rimossa al frame corrispondente a tale blocco.

Lo stato interno di questo programma è rappresentato in Figura 22.

> **Nota bene.** Potremmo programmare senza blocchi? Se non esistesse questo meccanismo di pile di frame, non si potrebbe dare lo stesso nome a più variabili all'interno dello stesso programma. Riusare nomi di variabili risulta comodo e 'naturale', si pensi ad esempio a quando vogliamo chiamare un contatore `i` o una variabile temporanea `tmp`. Questo avviene anche in matematica, infatti si definiscono le funzioni $f(x)$, $g(x)$, etc. nonostante ciascuna di queste x sia una "x" diversa.

3.6 Struttura di un programma in `C`

Introduciamo adesso la struttura "astratta" di un programma in `C`, così da poter iniziare a scrivere codici completi, funzionanti e sintatticamente corretti.

Il seguente frammento di codice rappresenta lo scheletro di programma in `C` (ovvero un template astratto), da memorizzare in un file `main.c`, ad esempio:

```c
/* vogliamo usare comandi definiti all'interno della standard library (stdio) */
#include <stdio.h> //inclusione librerie esterne

int main(void){
    /* stampare su schermo il messaggio "Hello world" */
    printf("Hello world\n"); //corpo della funzione main
    return 0;
}
```

La prima riga di codice, `#include<stdio.h>` consente di utilizzare comandi definiti nella *standard library* del linguaggio (in questo caso la funzione `printf`). Una libreria non è altro che un meccanismo per fornirci un insieme di funzionalità - cioè codice - senza che si debba riscrivere tutto; nell'ambito del corso di Laboratorio di programmazione vedrete come costruire moduli Python, un altro modo per creare delle "librerie". Nel programma qua sopra `main` è invece una *funzione* - dettagli su cosa siano e come si definiscono le

funzioni verranno dati in seguito. All'interno della funzione `main` definiamo il flusso di esecuzione principale del nostro programma `C`.

Compilare ed eseguire programmi. Compilare un programma significa tradurre il codice in `C` in un linguaggio di più basso livello (ovvero più comprensibile dal calcolatore) in modo che questo sia eseguibile. In ambienti come quelli disponibili su Repl.it queste operazioni sono spesso automatizzate tramite bottoni, quali il bottone RUN.

Se non si usano ambiente come Repl.it e si vuole gestire manualmente il processo di compilazione ed esecuzione, possiamo lavorare con una shell.

```
gcaravagna@guascone ~ % cd src
gcaravagna@guascone ~/src % gcc main.c -o my_executable_file
gcaravagna@guascone ~/src % ./my_executable_file
```

Qui sto usando una shell UNIX standard, dove il mio nome utente è `gcaravagna`, la macchina si chiama `guascone` e `%` indica il prompt dei comandi. Le operazioni che eseguo suppongono che `main.c` sia dentro alla cartella `src`:

1. entro nella cartella `src`, dove ho salvato il mio file `main.c`;
2. chiamo un compilatore (qui `gcc`) passando il file con il sorgente del programma, e chiedendo che l'output venga memorizzato in un nuovo file di nome `my_executable_file`;
3. eseguo il programma mediante l'esecuzione del file `my_executable_file` - .

Stampare variabili su schermo. Molte volte può essere utile stampare su schermo il valore delle variabili che stiamo utilizzando nel nostro codice, magari accompagnate da qualche commento. Questo può esser fatto tramite la funzione `printf` (che possiamo utilizzare solo dopo aver importato la standard library tramite `#include<stdio.h>`), ad esempio:

```c
#include <stdio.h>

/* vogliamo calcolare l'area di un triangolo */
int main(void){

    /* dichiarazioni variabili */
    int base, altezza;
    float area;

    /* inizializzazione variabili */
    base = 10;
    altezza = 34;

    /* stampo su schermo i miei dati, descrivendoli */
    printf("Base del triangolo: %d\n", base);
    printf("Altezza del triangolo: %d\n", altezza);
```

```
    /* calcolo e stampo il valore dell'area */
    area = (base * altezza) / 2;
    printf("Area del traingolo: %f\n", area);

    return 0;
}
```

In particolare, i caratteri tra apici denotano stringhe, `%d` significa che il valore che vogliamo stampare è un intero, mentre `%f` che è un float; in quest'ultimo caso è possibile stabilire quante cifre decimali visualizzare, ad esempio se ne vogliamo stampare 3 il comando è `printf("%.3f", valore);`.

Con il simbolo `\n` (newline) diciamo alla funzione di stampa di andare a capo.

Acquisire valori da tastiera. Nel programma dell'esempio precedente (così come in tutti quelli visti fino ad ora) le variabili vengono inizializzate con valori predefiniti (conosciuti a priori). Quindi, nel caso in cui si voglia eseguire lo stesso calcolo partendo da valori diversi occorre individuare nel codice il punto in cui le variabili vengono inizializzate e modificare 'a mano' il loro valore.

Sarebbe molto più comodo scrivere un codice generico, che esegue il calcolo che vogliamo per qualsiasi valore dei suoi input. Un modo per ottenere questo comportamento è acquisire i valori delle variabili da tastiera, ovvero chiedere all'utente durante l'esecuzione del programma quali valori devono assumere le variabili in input.

Questo si ottiene tramite la funzione `scanf`, anche questa parte della standard library, la cui sintassi è mostrata nel seguente frammento di codice:

```
#include<stdio.h>
/* vogliamo calcolare l'area di un triangolo */
int main(void){

    /* dichiarazioni variabili */
    int base, altezza;
    float area;

    /* acquisizione variabili con scanf */
    printf("Inserire valore base: ")
    scanf("%d", &base);
    printf("Inserire valore altezza: ")
    scanf("%d", &altezza);

    /* stampo su schermo i miei dati, descrivendoli */
    printf("Base del triangolo: %d\n", base);
    printf("Altezza del triangolo: %d\n", altezza);

    /* calcolo e stampo il valore dell'area */
    area = (base * altezza) / 2;
    printf("Area del traingolo: %f\n", area);
```

```
    return 0;
}
```

La sintassi di `scanf` è molto simile a quella di `printf`, in particolare con `%d` acquisiamo un valore intero (`int`) e con `%f` un `float`. E' importante ricordarsi di inserire la 'E commerciale' `&` prima del nome della variabile da acquisire (ne capiremo il motivo e il significato quando verranno spiegati i puntatori). Inoltre è necessario che le variabili il cui valore viene acquisito da tastiera siano state dichiarate (ovvero il loro tipo sia noto).

Esercizio 3.4. Si provi a compilare ed eseguire il suddetto programma C, seguendo le istruzioni riportate sopra per quanto riguarda l'utilizzo della shell.

Bonus tipi: `char`. Nonostante non sia uno degli argomenti rilevanti di queste lezioni, vogliamo introdurre il concetto di "carattere", che in `C` sono variabili di tipo `char`. La sintassi è pressoché la stessa che nel caso di variabili numeriche, come mostra il seguente frammento di codice:

```c
#include <stdio.h>

int main(void){

    char c; //variabile di tipo char
    int vocale_minuscola, vocale_maiuscola;

    /* acquisisco valore di c da tastiera */
    printf("Inserire input: ");
    scanf("%c", &c);

    /* predicati logici */
    vocale_minuscola = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
    vocale_maiuscola = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');

    /* voglio sapere se l'input inserito è o meno una vocale */
    if(vocale_minuscola || vocale_maiuscola)
        printf("Vocale\n");
    else
        printf("Non-vocale\n");
}
```

Quindi per stampare o acquisire da tastiera il valore di un `char` bisogna usare la sintassi `%c`.

4 Funzioni

4.1 Il concetto di funzione nella programmazione

Molte volte scriviamo (parti di) programmi che possono essere utilizzati più volte (e non vorremmo riscrivere ogni volta che ci servono) o che comunque descrivono una procedura a sé stante, che codifica una computazione ben precisa ed indipendente dal resto del codice che scriviamo. In questi (ed altri) casi conviene racchiudere tale porzione di codice all'interno di **funzioni**.

Definizione 4.1 (Funzione)**.** Una funzione calcola un valore, che può essere opzionalmente restituito. La sintassi per definire una funzione in `C` è la seguente:

```
tipo_ritorno nome_funzione(tipo_argomento_1 nome_argomento_1,
                           tipo_argomento_2 nome_argomento_2,
                           ...)
{
    corpo_funzione;
    return nome_output;
}
```

dove:

- `nome_funzione` è appunto il nome (arbitrario) che diamo alla nostra funzione (e.g., "radice", "quadrato", "primo");
- `corpo_funzione` è il codice che determina il comportamento della funzione (che può ovviamente contenere istruzioni condizionali, iterative, etc.);
- `tipo_ritorno` è il tipo del valore che verrà restituito come output dal calcolo della funzione;
- `nome_output` è il nome della variabile - o dell'espressione - utilizzata per determinare il valore restituito dalla funzione;
- `tipo_argomento` e `nome_argomento` sono rispettivamente il tipo ad il nome delle variabili che la funzione prende in input (gli input di una funzione sono chiamati *argomenti*).

La *segnatura* (o firma) di una funzione è costituita dal tipo di ritorno, il nome della funzione e il tipo degli argomenti.

Nota bene (Relazioni con le funzioni matematiche)**.** In matematica definiamo

$$f(x) = x^2 + \log(x)$$

con $f : \mathbb{R} \to \mathbb{R}$. Questa definizione è perfettamente consistente con il concetto di funzione appena descritto:

- f è il nome della funzione
- il tipo di ritorno è un valore dell'insieme $\mathbb{R}$, il codominio di f;
- il tipo di input è un valore dell'insieme $\mathbb{R}$, il dominio di f;

- il corpo della funzione f è l'espressione algebrica $x^2 + \log(x)$

Si noti che la definizione di f utilizza un'altra funzione, log. `void` e non è obbligatorio inserire un comando `return`.

Nota bene (Funzioni void in output)**.** Una funzione potrebbe non restituire un valore - e.g., una funzione che stampa un semplice messaggio a schermo, o semplicemente modifica valori di variabili. In tal caso si utilizza un tipo di ritorno speciale, denotato con il termine `void`. Per tale funzione non è obbligatorio inserire un comando `return`.

Nota bene (Funzioni void in input)**.** Una funzione che non richiede parametri può ricevere un parametro `void` in input, oppure non avere nessun parametro esplicitato.

Esercizio 4.1. Con riferimento ai semplici programmi descritti sopra, identificare le componenti della firma di ciascuna funzione `main`. Si noti che alcune volte le funzioni non esplicitano il `return`, nonostante debbano ritornare un valore; in tal caso si ritorna implicitamente il risultato dell'ultima istruzione eseguita (e.g., assegnamento).

Possiamo cominciare con la definizione di una funzione che riprenderemo spesso in questa dispensa.

Esempio 4.1 (Calcolo dell'MCD con una funzione)**.** Nell'Esempio 3.7 abbiamo implementato una procedura per il calcolare il MCD utilizzando l'algoritmo di Euclide.
In questo punto del corso, questa stessa procedura può esser scritta come funzione nel seguente modo:

```
/* dichiarazione di funzione in C */
void mcd(int p, int s)
{
    while(p != s)
    {
        if(p>s)
            p = p - s;
        else
            s = s - p;
    }
}
```

Da notare che la funzione ha come tipo di ritorno `void` poiché il valore del MCD è contenuto nelle variabili `p, s` che vengono modificate durante l'esecuzione della funzione `mcd`. Questa scelta verrà dibattuta fra poco.

4.2 Passaggio dei parametri, e semantica dei blocchi

Il **passaggio di parametri** è il collegamento tra argomenti *formali* di una funzione (quelli specificati in fase di definizione) e parametri (detti *attuali*), ovvero variabili che vivono all'interno del nostro programma e che vogliamo

usare come input per la funzione che abbiamo definito. Per esempio, in matematica definisco $f(x)$, e poi in un calcolo specifico uso $f(6)$, ovvero lego la variabile x al valore 6.

In `C` il passaggio dei parametri avviene solamente *per valore*; quando introdurremo la nozione di puntatore, vedremo che i parametri possono esser usato per simulare anche un passaggio per riferimento, ma rimarrà comunque vero il fatto che in `C` esiste solamente il passaggio per valore.

Consideriamo la funzione definita nell'Esempio 4.1, possiamo chiamarla (ovvero passarle dei parametri per far sì che calcoli su determinati input ciò che abbiamo definito al suo interno) nel seguente modo:

```
/* .... */
{
    int x = 30;
    int y = 12;
    /* chiamata di funzione */
    mcd(x, y); //calcolo mcd tra 30 e 12
}
```

Parametri, funzioni e memoria. Come vengono eseguite le funzioni? Semplicemente, come i blocchi, perché il corpo della funzione altro non è che un blocco di codice. Quindi, viene eseguito in un suo nuovo frame, che viene distrutto alla fine dell'esecuzione del blocco stesso.

Per semplicità di comprensione, possiamo immaginare che:

- sia ρ e σ la coppia ambiente/memoria prima della chiamata di funzione;
- un nuovo frame viene creato sopra ρ e σ, e riempito con le dichiarazioni dei parametri della funzione;
- un ulteriore nuovo frame viene creato ed utilizzato per gestire il blocco di codice che definisce la funzione;
- alla terminazione dell'esecuzione della funzione entrambi i frame sono eliminati.

Esercizio 4.2. Si ragioni sul concetto qui sopra espresso, e si veda come questo possa concettualmente permetter di definire i seguenti programmi

```
// file main.c, con main omesso
int test(int x)
{
    int x = 30;
    return(x);
}
```

Quanto vale il valore di ritorno di questa funzione? Teoricamente dovrebbe valere 30, anche se ovviamente la funzione in se non ha senso perchè dichiara

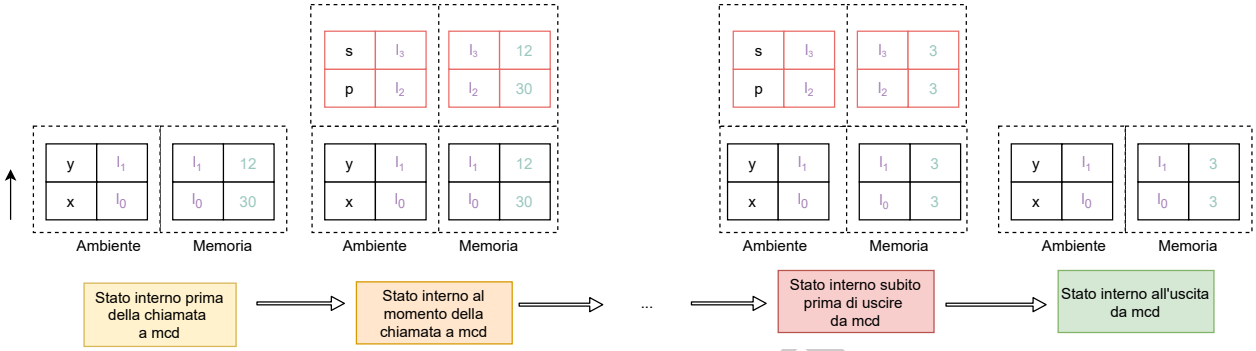


Figura 23. Modifiche alla pila di frame durante la chiamata alla funzione `mcd` dell'Esempio 4.1.

un parametro che non è mai utilizzabile.

Ad ogni modo, in alcune implementazioni del `C` funzioni come quelle sopra non sono permesse, nel senso che in fase di compilazione sono sollevati errori. Su Repl.it ad esempio:

```
~/test$ gcc main.c
main.c: In function ‘test’:
main.c:4:9: error: ‘x’ redeclared as different kind of symbol
     int x;
         ^
main.c:3:14: note: previous definition of ‘x’ was here
 int test(int x){
```

> **Nota bene.** All'interno della definizione di una funzione possiamo chiamare altre funzioni! Quando una funzione viene chiamata all'interno di un'altra funzione, essendo una funzione di per sé un blocco, un nuovo frame viene aggiunto agli stack di ambiente e memoria, che verrà poi rimosso una volta che la funzione ritorna, ovvero conclude la sua esecuzione. In Figura 23 questo meccanismo è mostrato schematicamente per la chiamata alla funzione `mcd` descritta sopra.

Ragioniamo ora della funzione MCD che abbiamo definito ed eseguito in Figura 23. Quella definizione non ha particolarmente senso, semplicemente perché non restituendo un valore, non abbiamo modo di "propagare" (leggasi restituire) il calcolo dell'MCD a chi esegue la funzione. Miglioriamo quindi la definizione.

> **Esempio 4.2** (MCD con valore di ritorno)**.** Consideriamo ancora il calcolo del MCD tramite l'algoritmo di Euclide, come nell'Esempio 4.1. Nella precedente implementazione, la funzione `mcd` non aveva un valore di ritorno, ma modificava solamente i suoi argomenti. Possiamo ritornare un intero corrispondente al MCD tra i due input nel seguente modo:
> ```
> int mcd(int p, int s)
> ```

```
{
    while(p != s)
    {
        if(p>s)
            p = p - s;
        else
            s = s - p;
    }

    /* ritorno un intero che contiene il MCD */
    return p; //oppure return s, essendo uguali per costruzione;
}

/* utilizzo di questa funzione in un main */
int main(void)
{
    int x = 30;
    int y = 12;

    /* dichiaro una variabile che contiene il MCD */
    int z = mcd(x, y); // esecuzione della funzione per valori di x ed y
}
```

Ovvero, la variabile `z` dentro la quale vogliamo memorizzare il MCD viene dichiarata con tipo `int` ed inizializzata tramite la chiamata alla funzione `mcd`.

Possiamo cominciare quindi a definire le funzioni che ci interessano, e che ci interesseranno anche per introdurre altri concetti. Riprendiamo il *fattoriale* di un numero $n > 0$ intero, ovvero:

$$n! = \begin{cases} 1 & \text{se } n \in \{0, 1\} \\ n \cdot (n-1)! & \text{altrimenti} \end{cases}$$

Esempio 4.3 (Funzione fattoriale)**.** Consideriamo, come nell'Esempio 3.6, il calcolo del fattoriale di un numero intero. Questa procedura può essere implementata come funzione nel seguente modo:

```
/* funzione che calcola il fattoriale */
int fact(int n)
{
    int f = 1 // variabile che verrà ritornata

    while(n>1)
    {
        f = f * n;
        n = n - 1;
    }

    return f;
```

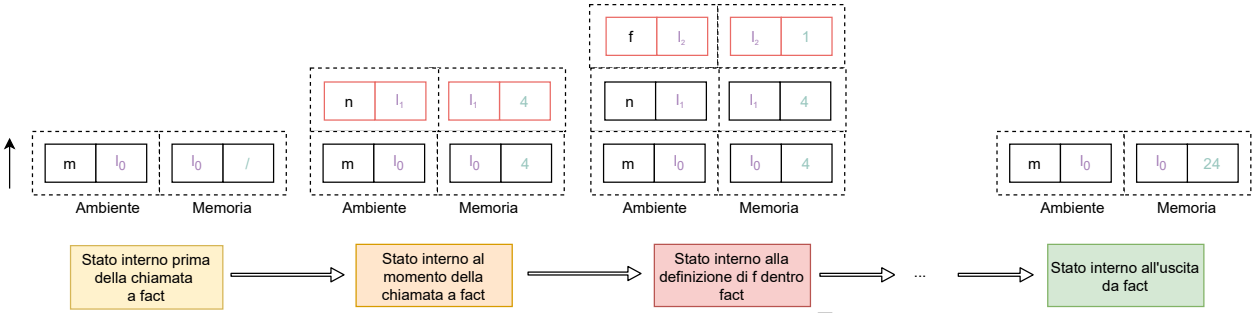


Figura 24. Stato interno del programma dell'Esempio 4.3. Da notare che viene aggiunto un frame al momento della chiamata alla funzione `fact` contenente indirizzo e valore dei suoi parametri e un altro frame al momento della dichiarazione di variabili all'interno della funzione. Entrambi questi frame vengono eliminati dallo stack nel momento in cui la funzione ritorna.

```
}

/* utilizzo */
int main()
{
    int m = fact(4);
}
```

Dal momento che il fattoriale di un numero intero è un numero intero, il tipo di ritorno della funzione `fact` è `int`. In Figura 24 è mostrata l'evoluzione dello stato interno del programma durante l'esecuzione di questo codice.

Nota bene. E' importante commentare bene le funzioni, ovvero spiegare cosa si vuole calcolare con la funzione che stiamo definendo e cosa rappresentano gli input. Ad esempio:

```
/*
    Funzione per calcolare il fattoriale.
    Argomenti:
    x: intero. Numero di cui si vuole calcolare il fattoriale.
    Output
    int: fattoriale di x.
*/
int fact(int x)
{
    ...
}
```

In particolare ci sono due simboli che possono esser usati per scrivere commenti all'interno di un programma in `C`, finora utilizzati senza darvi opportune spiegazioni: `/* ... */` rappresenta un commento che può occupare più righe, `//...` rappresenta un commento su una sola riga.

4.3 Funzioni ricorsive

Una funzione si dice **ricorsiva** quando all'interno della sua definizione contiene una chiamata a sé stessa. Oltre alla chiamata a sé stessa, per sapere quando concludere l'esecuzione, occorre all'interno di una funzione ricorsiva definire un caso base la cui soluzione è nota.

Consideriamo il calcolo del fattoriale visto nell'Esempio 4.3, dal momento che vale:

$$n! = \begin{cases} 1 & \text{se } n \in \{0, 1\} \\ n \cdot (n-1)! & \text{altrimenti} \end{cases}$$

possiamo implementare la definizione ricorsiva della funzione `fact` nel seguente modo:

```
int fact_rec(int n)
{
    if(n<=1)
        // caso base della ricorsione
        return 1;
    else
        // chiamata ricorsiva
        return n*fact_rec(n-1);
}
```

Si noti che `fact_rec` contiene effettivamente la chiamata ricorsiva a se stesso, precisamente usando `fact_rec` con argomento `n-1`. Il caso speciale `n<=1`, corrispondente a $n \in \{0, 1\}$ nella definizione matematica assumendo non vi siano input negativi, si chiama *caso base* della ricorsione.

Lo stato interno del programma `fact_rec` è mostrato schematicamente in Figura 25.

Il pattern di programmazione ricorsiva è sempre

```
... f(x, ...)
{
    if(caso base)
        // caso base della ricorsione
        return valore_caso_base;
    else
        // chiamata ricorsiva
        return ... f(y);
}
```

dove `y` è un valore più piccolo (o grande) di `x` - è corretto dire che sia sicuramente diverso da `x`, altrimenti la definizione non sarebbe ben fondata perchè darebbe luogo ad una definizione infinita.

> **Nota bene** (Ricorsione ed induzione, due faccie della stessa medaglia)**.** Matematicamente, per poter scrivere una funzione ricorsiva su un dominio di valori - e.g., nel fattoriale i valori sono 0, 1, 2, ... etc. - deve valere che

il dominio stesso è costruibile per induzione. Infatti i numeri naturali sono definibile per induzione come segue: 0 è un numero naturale, ed il successore di un numero naturale è un numero naturale. Ed i numeri reali?

Esempio 4.4 (Somma delle cifre di un numero)**.** Dato un numero $c_1c_2 \dots c_n$, calcolare ricorsivamente la somma delle cifre $c_1+c_2+\dots+c_n = \sum_{i=1}^{n} c_i$, ad esempio dato il numero 2386 si vuole calcolare $2+3+8+6$. Questo può essere risolto nel seguente modo:

```
int somma_cifre(int n)
{
    /* caso base: numero di una sola cifra */
    if(n==0)
        return 0;

    /* bisogna estrarre cifra per cifra, usiamo il modulo */
    double f = n % 10; //ultima cifra
    double g = n / 10; //numero senza l'ultima cifra

    return f + somma_cifre(g);
}
```

Si ragioni del perché il caso base sia 0 e non un numero qualunque di una singola cifra. Si provi inoltre ad eseguire carta e penna questa funzione.

Esercizio 4.3. Si disegni la memoria di un programma che calcola `somma_cifre(123)`, usando la definizione dell'esempio 4.4.

Esempio 4.5. Dati due numeri interi $a > 0, b > 0$, visualizzare con `printf` l'intervallo $[a, b]$ (assumendo $a < b$) stampando con una procedura ricorsiva i numeri pari o dispari nell'intervallo in base al valore iniziale di a.

Ad esempio se $a = 1, b = 5$ stampare $1, 3, 5$ invece se $a = 2, b = 10$ stampare $2, 4, 6, 8, 10$. Una possibile soluzione è data dal seguente codice:

```
void stampa_intervallo(int a, int b)
{
    /* la computazione è definita solo se a<=b */
    if(a<=b)
    {
        /* stampo a */
        printf("%d", a);

        /* chiamata ricorsiva */
        stampa_intervallo(a+2, b); //sposto a di due interi
    }
}
```

Esercizio 4.4. Si potrebbero scrivere la funzione degli esempi 4.4 e 4.5 in modo iterativo?

Esercizio 4.5. Si consideri il programma dell'Esempio 4.5, modificare il programma in modo che dato un parametro δ e due interi $a > 0, b > 0$ stampi $a, a+\delta, a+2\delta, \ldots$ all'interno dell'intervallo $[a, b]$, usando una procedura ricorsiva.

4.4 Funzioni iterative e ricorsive per il calcolo di serie e successioni

Come probabilmente già sapete, in analisi matematica una successione è una particolare funzione definita sull'insieme dei numeri naturali e con valori nell'insieme dei numeri reali (ovvero è una sequenza ordinata di numeri reali).

Consideriamo ad esempio la seguente successione, definita come:

$$s_n = \begin{cases} 4 & n = 0 \\ 4 * s_{n-1} + 3 * n & n > 0 \end{cases} \tag{4.1}$$

Nel gergo delle successioni, n prende il nome di *indice della successione*, mentre s_n è chiamato n-esimo termine della successione.

Si può osservare che questa successione è definita ricorsivamente (o per ricorrenza), in quanto si ottiene specificando un caso base (in questo caso $n = 0$) e una funzione tale che $s_n = f(s_{n-1})$.

Il nostro obiettivo è scrivere un programma in `C` che calcoli l'n-esimo termine di questa successione (dove n è un parametro in input), in modo ricorsivo.

Una possibile soluzione è la seguente:

```
int succ_rec(int n){
    //caso base
    if(n==0)
        return 4;
    //chiamata ricorsiva
    else
        return 4*succ_rec(n-1) + 3*n;
}
```

Possiamo naturalmente anche scrivere una funzione iterativa che calcoli l'n-esimo termine di questa successione:

```
int succ_iter(int n){
    //caso base (ovvero primo termine)
    int s_n_1 = 4; //tengo traccia dell'(n-1)-esimo termine
    //calcolo termine i-esimo come s_i = f(s_i-1)
    for(int i=1; i<=n; i++)
```

```
    {
        //aggiorno n-esimo termine
        int s_n = 4*s_n_1 + 3*n;
        //aggiorno (n-1)-esimo termine
        s_n_1 = s_n;
    }
    //ritorno n-esimo termine
    return s_n;

}
```

Da questo semplice esempio possiamo ricavare dei pattern di programmazione da utilizzare nei casi in cui ci venga richiesto di trovare il termine n-esimo di una successione, sia ricorsivamente che iterativamente.

Nel caso ricorsivo il pattern è il seguente:

```
int successione_ricorsiva(int n)
{
    //definire caso base
    if(n==caso_base)
        return valore_caso_base;
    //chiamata ricorsiva, chiamata a sè stessa più eventuali altre operazioni (racchiuse in una funzione f)
    else
        return f(successione_ricorsiva(n-1));
}
```

Nel caso iterativo il pattern è il seguente:

```
int successione_iterativa(int n)
{
    //definire primo termine (dato dal caso base)
    int s_n_1 = valore_caso_base;
    //fare un ciclo for da caso_base+1 a n
    for(int i = caso_base+1; i<=n; i++)
    {
        //calcolare termine i-esimo della funzione come funzione del termine (i-1)-esimo
        int s_n = f(s_n_1);
        //aggiornare termine (i-1)-esimo
        s_n_1 = s_n;
    }
    //ritornare ultimo termine calcolato
    return s_n;
}
```

Funzioni e principio di modularità. La chiamata di una funzione sospende la procedura (ovvero la funzione) correntemente in esecuzione - si noti per l'appunto che l'esecuzione di un programma `C` comincia sempre con l'esecuzione della funzione `main` associata.

Ad esempio noi sappiamo che quando eseguiamo un codice in `C` siamo all'interno della funzione `main`: supponiamo che venga chiamata la funzione

`mcd`, cosa succede? `main` viene interrotta ed andiamo all'interno di `mcd`, in un nuovo frame nell'ambiante e nella memoria.

Quando la procedura chiamata termina, la funzione che é stata interrotta riprende la sua esecuzione. Continuando nel nostro esempio, quando la funzione `mcd` termina, la funzione `main` riprende la sua esecuzione, ovvero il frame di `mcd` viene eliminato.

Il **principio di modularità** - che é una buona pratica in programmazione - consiste nell'implementare come funzione tutti quei frammenti di codice che hanno senso di per sé. Il codice viene quindi spezzato in componenti (moduli) separati, e questo rende un programma più leggibile e più facile da aggiornare ed estendere (o rimpiazzare, per esempio, sostituendo solamente alcuni dei suoi moduli/ funzioni).

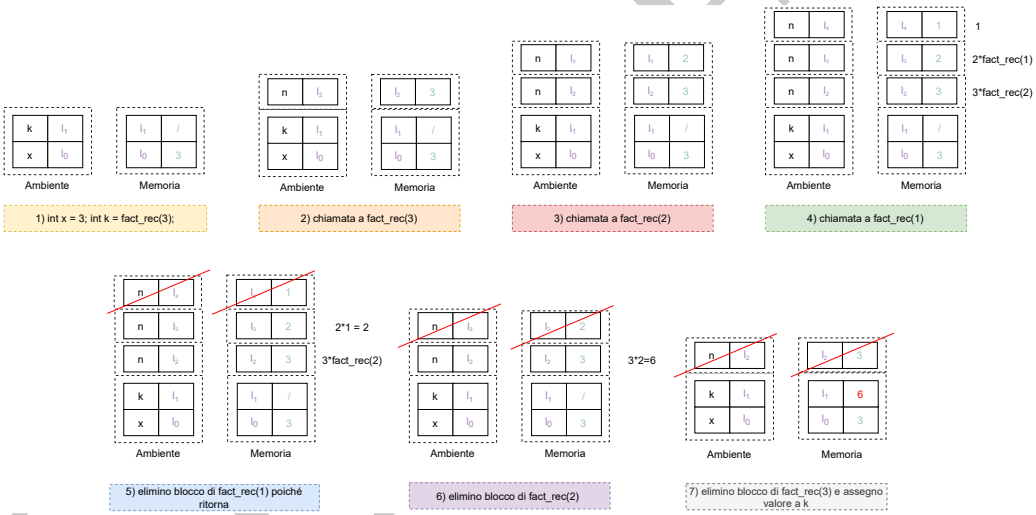


Figura 25. Stato interno del programma `fact_rec`. Fino a che la funzione ricorsiva non incontra il caso base continua a chiamare sé stessa, aggiungendo un blocco sul frame di ambiente e memoria. Questi blocchi sono poi eliminati nell'ordine inverso rispetto a quello in cui sono stati creati (LIFO policy) non appena la funzione arriva al caso base.

5 Puntatori

5.1 Definizione e modello in memoria

Molte volte capita che tramite una funzione si vogliono modificare i valori di più variabili contemporaneamente, ma, ritornando la funzione un solo valore (o come vedremo quando parleremo di array, un insieme di valori tutti dello stesso tipo), questo non ci è possibile con gli strumenti visti fin'ora.

Un altro caso in cui non ci sono sufficienti i costrutti visti fino ad ora è quello in cui vogliamo modificare permanentemente gli argomenti di una funzione, e ciò non è possibile se questi vengono passati per valore.

Possiamo motivare concretamente queste considerazioni con un esempio di una funzione che scambia i valori di due variabili ricevute in input. Tale funzione vorrebbe essere usata per modificare il calcolo dell'MCD.

Esempio 5.1 (Scambio di valori)**.** Per calcolare l'MCD tra x ed y tramite l'algoritmo di Euclide, possiamo:

- garantire che x contenga sempre il massimo tra x ed y;
- ad ogni iterazione calcolare $x - y$, eliminando così un ramo condizionale.

Supponiamo di aver definito una funzione `scambia` che presi x ed y scambia il loro valore, ad esempio se $x = 4, y = 5$ si ha che `scambia`(x, y) produca una memoria dove $x = 5, y = 4$.

In codice la nuova procedura per calcolare il MCD diventerebbe dunque:

```
int mcd(int x, int y)
{
    while(x!=y)
    {
        /* vogliamo che x contenga sempre il max(x, y) */
        if(y>x)
            scambia(x,y);

        /* algoritmo di Euclide */
        x = x - y;
    }

    /* x contiene il MCd tra x e y */
    return x;
}
```

Nonostante il compito di scambiare due valori contenuti in due variabili distinte possa sembrare banale, gli strumenti visti fino ad ora non sono sufficienti a eseguire questa computazione.

Esercizio 5.1. Si consideri

```c
void scambia(int p, int q)
{
    p = q;
    q = p;
}
```

Disegnare la memoria per l'esecuzione di questa funzione, e commentare la sua correttezza.

Proseguiamo nel nostro tentativo per mostrare (in pratica) le limitazioni dei programmi che sappiamo scrivere fino ad ora.

> **Nota bene** (I frame e la loro vita (breve)). Definiamo un'implementazione di `scambia` con variabile d'appoggio:
> ```c
> void scambia(int p, int q)
> {
>     // variabile d'appoggio
>     int tmp;
>
>     /* supponiamo p=5, q=1 */
>     tmp = p; // tmp=5
>     p = q;   // p=q=1
>     q = tmp; // q=tmp=5
> }
>
> /* chiamata di funzione */
> int main()
> {
>     int x = 5;
>     int y = 1;
>
>     scambia(x, y);
> }
> ```
> Quello che succede a livello di ambiente e di memoria durante l'esecuzione di questa funzione è mostrato in Figura 26.
>
> Ovvero, la funzione `scambia` fa effettivamente quello che chiediamo, però all'interno del suo frame, quindi tutte le modifiche sono perse una volta che la funzione termina la sua esecuzione ed il suo frame viene eliminato!

Come possiamo garantirci che le modifiche si ripercuotano al di fuori del frame locale in cui esse avvengono?

Quello che serve per implementare correttamente la funzione `scambia` sono i **puntatori**, che il `C` ci fornisce come un tipo primitivo del linguaggio.

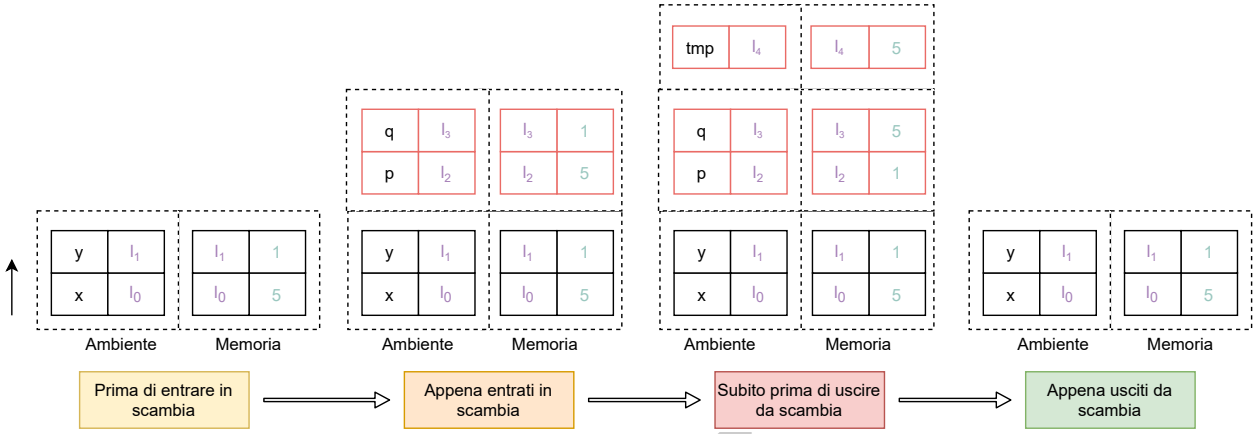


Figura 26. Stato del programma durante l'esecuzione della funzione `scambia` implementata utilizzando una variabile d'appoggio.

Definizione 5.1 (Puntatore)**.** Un **puntatore** è una variabile il cui valore è un *indirizzo di memoria* (o locazione). Si noti quindi che, essendo `C` un linguaggio fortemente tipato, anche i puntatori sono associati ad un tipo. La sintassi per la definizione di un puntatore è quindi diversa da quella di una variabile normale:

```
tipo * nome_puntatore;
```

ovvero dopo al nome del tipo va aggiunto un asterisco, ad esempio `int *` o `double *` per indicare un puntatore ad intero, o una double.

Contestualmente alla dichiarazione del puntatore, possiamo introdurre il simbolo `&` - chiamato *address of*, o *ampersand* - utilizzato per accedere all'indirizzo di una variabile.

Il costrutto di ampersand serve per assegnare un puntatore ad una variabile, che deve essere del tipo opportuno rispetto al puntatore, ad esempio

```
int x = 10; // intero
int* p;     // puntatore a un intero

p = &x;     // assegno a p l'indirizzo di x
```

Lo stato di questo programma è rappresentato in Figura 27. In particolare la variabile `p` contiene un indirizzo di memoria, nello specifico l'indirizzo di memoria in cui si trova l'intero `x`.

Per modificare il valore di una variabile puntata da un puntatore bisogna usare l'operazione di *dereferenziazione* (dereferencing, denotato `*p`), che permettere di accedere al valore della variabile puntata dal puntatore. La sintassi è la seguente:

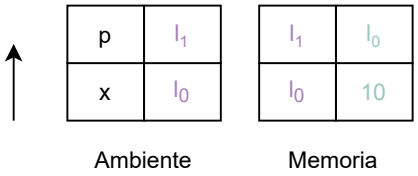


Figura 27. Rappresentazione di ambiente e memoria in presenza di un puntatore. Notare come il valore memorizzato associato a `p` sia un indirizzo, e non un numero o un carattere etc. come nel caso delle altre variabili.

```
int x = 10;
int y = 20;

int* p = &x; //assegno a p l'indirizzo di x
int* q = &y; //assegno a q l'indirizzo di y

/* Effettuo due operazioni equivalenti */

x = y + 1; // primo metodo, modificando i valori delle variabili

*p = *q + 1; // secondo metodo, usando le proprietà dei puntatori
```

Si noti quindi che `*p` ha due significati diversi a seconda del fatto che sia usato al momento della dichiarazione, o della dereferenziazione.

5.2 Passaggio di parametri per indirizzo

I puntatori possono essere anche argomenti di funzioni, fornendo il così detto passaggio dei parametri per "indirizzo" (detto anche *per riferimento*).

> **Nota bene** (Passaggio per riferimento versus per valore)**.** In `C` esiste solo il passaggio per valore. Essendo però i puntatori - e quindi gli indirizzi - un possibile valore, otteniamo in pratica il passaggio per riferimento.

Passando ad una funzione l'indirizzo di memoria in cui si trova la variabile che intendiamo modificare (anziché il valore da manipolare) ogni modifica a quell'indirizzo di memoria permane anche dopo il completamento dell'esecuzione della funzione.

> **Esempio 5.2** (Funzione `scambia` corretta)**.** Come nell'Esempio 5.1, vogliamo implementare una funziona che scambia il valore di due variabili.
>
> Utilizzando i puntatori, riusciamo ad implementarla in modo corretto:
>
> ```
> // void, perché non deve restituire alcun valore
> void scambia(int* p, int* q)
> {
>     int tmp = *p; // tmp contiene il valore all'indirizzo p
> ```

```
    /* il valore all'indirizzo p diventa uguale a quello all'indirizzo q */
    *p = *q;

    /* il valore all'indirizzo q diventa uguale a tmp */
    *q = tmp;
}
```

Il concetto di modificare, all'interno di un blocco che verrà distrutto, zone di memoria create in un blocco creato precedentemente, è molto potente.

Esempio 5.3 (Modifiche persistenti)**.** Vogliamo scrivere una funzione che incrementi il valore di una variabile, utilizzando i puntatori, ovvero quello che vogliamo ottenere è il seguente comportamento:

```
int main()
{
    int x = 0;
    int y = 5;

    incr(x, 5); //aumenta x di 5
    incr(y, 7); //aumenta y di 7
}
```

Ovvero la funzione `incr` che vogliamo definire aumenta il valore di una certa variabile (`x`, che passeremo in input) di un certo valore (`y`, anch'esso passato in input).

Dal momento che la modifica al valore della variabile deve essere persistente, la variabile il cui valore vogliamo cambiare va passata tramite puntatore. Possiamo usare questo codice:

```
void incr(int* p, int q)
{
    /* dereferenziazione */
    *p = *p + q;
}

/* chiamata nel main */
int main()
{
    int x = 0;
    int y = 5;

    /* dal momento che il primo argomento è un puntatore, occorre usare & */
    incr(&x, 5); //aumenta x di 5
    incr(&y, 7); //aumenta y di 7
}
```

Esercizio 5.2. Si disegni la memoria del programma nell'Esempio 5.3, con-

vincendosi della persistenza delle modifiche ad `x` e `y` nel `main`.

Esempio 5.4 (La funzione scanf)**.** Come abbiamo già menzionato nella Sezione 3.6, la funzione `scanf` prende come secondo argomento l'indirizzo del nome della variabile di cui si vuole leggere il valore da tastiera. Ad esempio per leggere un intero `x` da tastiera occorre fare `scanf("%d", &x)`.
Adesso possiamo capire a fondo perché: noi vogliamo che la modifica ad `x` (ovvero il valore che gli diamo da tastiera) sia persistente, cioè sia tale anche nella porzione di programma successiva alla chiamata a `scanf`, e il modo corretto per ottenere questo comportamento è passare i parametri per indirizzo!

I puntatori possono essere utili anche quando vogliamo definire una funzione che calcola più di una quantità, dal momento che in `C` non è possibile ritornare più di un valore. :

Esempio 5.5 (Ritorno di valore doppio)**.** Definiamo una funzione che calcoli sia la somma che il prodotto di due interi, ovvero che dati x ed y resitutisca la coppia $(x+y, xy)$.

Possiamo ragionare di restituire un valore con il `return`, ed uno tramite un puntatore (la cui memoria puntata dovrà essere gestita dal chiamante).

```
int sommaprodotto(int x, int y, int* s)
{
    /* s contiene l'indirizzo contenente la somma */
    *s = x + y;

    /* il prodotto è il nostro output */
    return x*y;
}
```

Esercizio 5.3. Si disegni la memoria della funzione `sommaprodotto`.

6 Vettori

I vettori, in programmazione chiamati **array**, sono una struttura dati per rappresentare una sequenza finita di valori, in cui si tiene conto dell'ordine in cui sono memorizzati i dati. Nel contesto del lungaggui `C`, gli array devono anche contenere elementi tutti dello stesso tipo.

Cominciamo con il motivare la necessità pratica di avere gli array in un linguaggio di programmazione.

Esempio 6.1. Si vogliono sommare 5 numeri dati in input dall'utente. Con gli strumenti presentati fino ad ora la possibile soluzione sarebbe questa.

```c
#include<stdio.h> //printf, scanf
int main()
{
    // dichiaro 5 variabili
    int x, y, z, w, r;
    int tot;

    // Ottengo 5 valori dall'input, li memorizzo
    scanf("%d", &x);
    scanf("%d", &y);
    scanf("%d", &z);
    scanf("%d", &w);
    scanf("%d", &r);

    // Calcolo
    tot = x + y + z + w + r;
    printf("%d", tot);
}
```

Questo codice è poco elegante dato che molte righe sono replicate, e poco efficiente perché abbiamo bisogno di tante variabili. Ancora peggio, se decidessimo di richiedere 7 invece che 5 valori, dovremmo aggiungere 2 variabili e due chiamate a `scanf`.

L'utilizzo di array serve per gestire situazioni come queste. Cominciamo con il definire il tipo array.

Definizione 6.1 (Array)**.** Un **array** è una struttura di dati che descrive una sequenza di valori memorizzati, tutti dello stesso tipo.

Ogni elemento della sequenza è associato ad una posizione (*indice*), tramite la quale possiamo accedere e leggere o modificare il valore opportuno. La sintassi per dichiarare un array, per esempio di 10

Ad esempio parleremo spesso di "array di interi", o "array di float".

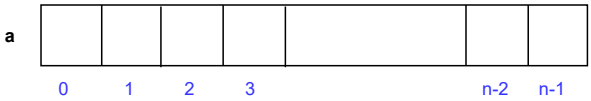


Figura 28. L'indicizzazione di un array inizia con l'indice 0. Ovvero, il primo elemento dell'array **a** (contenente n elementi) ha indice 0, il secondo 1, e così via, fino all'ultimo elemento che ha indice $n - 1$.

elementi è la seguente:

```c
/* dichiarazione array di 10 interi */
int a[10];

/* dichiarazione array di 10 float */
float b[10];

/* dichiarazione array di 10 double */
double c[10];

/* dichiarazione array di 10 caratteri */
char d[10];
```

Nota bene (Indicizzazione da 0)**.** Gli array si enumerano a partire da 0, come mostrato in Figura 28, quindi un array di $n = 10$ elementi si accede tramite le posizioni $0, \ldots, 9$.

Nota bene (Dichiarazione)**.** Gli array si dichiarano di una dimensione fissata, almeno per quanto abbiamo visto fino ad ora, e non si possono dichiarare usando codice[a] di questo tipo.

```c
/* richiesta di input */
int x;
scanf("%d", &x);

/* array di x elementi */
int a[x];
```

Per realizzare queste funzionalità avremo bisogno di concetti di memoria dinamica che vedremo nelle prossime sezioni.

[a] In realtà si riesce a farlo, ma solamente se `x` è un valore piccolo. Il problema è che questo tipo di codice denota una lacuna nella comprensione del funzionamento della memoria in `C`.

6.1 Lettura e scrittura di valori in un array

Per leggere i valori all'interno di un array `a` la sintassi è `a[...]` dove `...` denota l'indice al quale si vole accedere. Per esempio

```c
/* dichiarazione array */
int a[10];
```

```c
/* accedo all'elemento con indice 2, e sommo 3 */
int x = 3 + a[2];

/* accedo al primo elemento */
int y = a[0];

/* accedo all'ultimo elemento, e sommo 4 */
int z = 4 + a[9]; //l'array ha 10 elementi, quindi l'ultimo ha indice 9
```

Il modo per scrivere nelle varie posizioni degli array è molto simile alla lettura, almeno a livello di sintassi:

```c
/* scrivo nella cella corrispondente al primo elemento */
a[0] = 5;

/* scrivo nella cella corrispondente all'ultimo elemento */
a[9] = 7;

/* scrivo nella cella dell'i-esimo elemento */
int i = 4; //ok perché i è intero
a[i] = 8;
```

Inizializzazione di un array. Supponiamo di voler inizializzare un array con valore costante 0. A prima vista si potrebbe pensarlo di farlo nel seguente modo:

```c
/* dichiarazione array di 3 elementi */
int a[3];

/* inizializzo elemento per elemento */
a[0] = 0;
a[1] = 0;
a[2] = 0;
```

Tuttavia questo codice non è elegante (come prima, diverse righe sono più o meno ripetute) e richiede molto lavoro manuale per poter essere generalizzato ad array di dimensione diversa. Un modo più semplice e corretto di fare la stessa inizializzazione è utilizzando un ciclo `for`:

```c
/* dichiarazione array di 3 elementi */
int a[3];

/* inizializzazione con ciclo for */
for(int i=0; i<3; i++)
{
    a[i] = 0;
}
```

> **Nota bene.** Quando si accede ad un array bisogna sempre stare attenti alla sua dimensione. Ad esempio se un array `a` ha n elementi e si tenta di scrivere nella cella `a[n]` (ovvero un indice che non appartiene all'array), si va a scrivere al di fuori della memoria allocata per l'array. Si provi a

creare un programma `C` che genera questo errore.

Esempio 6.2 (Cancella negativi)**.** Vogliamo scrivere un programma che dato un array di interi `a`, lo modifichi in modo che tutti gli elementi negativi vengano sostituiti da 0. Ne diamo una parte omettendo textttmain e altre funzioni.

```
/* dichiarazione array */
int a[7];

/* inizializzazione, i.e. con qualche valore negativo */
a[0] = 2;
a[1] = -1;
a[2] = 0;
a[3] = 1;
a[4] = -1;
a[5] = 0;
a[6] = -3;

/* assegno 0 alle celle contenenti un numero negativo */
for (int i=0; i<7; i++)
{
    /* se i corrisponde ad un indice di un elemento negativo */
    if(a[i]<0)
        /* il contenuto della i-esima cella viene messo a 0 */
        a[i] = 0;
}
```

Esempio 6.3 (Incrementa pari)**.** Vogliamo scrivere un programma che incrementi di 1 tutti gli elementi in posizione pari di un array. Una possibile soluzione è:

```
/* dichiarazione e inizializzazione array */
int a[9];


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

/* iterazione sugli indici dell'array */
for(int i=0; i<9; i++)
{
    /* se l'elemento i è in posizione pari */
    if(i%2 == 0)
```

Esiste una sintassi alternativa per inizializzare un array: `int a[3] = 1,2,3;` crea un array `a` con gli elementi 1, 2 e 3.

```
        a[i] = a[i] + 1; //incremento il suo valore
}
```

Scriviamo adesso un programma completo che calcoli la somma degli elementi memorizzati in un array.

Esempio 6.4 (Somma elementi).

```
#include <stdio.h>
int main()
{
    int a[5];

    int tot = 0; //var. per memorizzare la somma

    /* chiedo in input gli elementi dell'array */
    for(int i=0; i<5; i++)
        /* inizializzazione i-esimo elemento */
        scanf("%d", &a[i]);

    /* calcolo somma totale elementi array */
    for(int i=0; i<5; i++)
        tot = tot + a[i];

    /* stampo il risultato */
    printf("Somma totale elementi array: %d",tot);
}
```

6.2 Array, memoria e puntatori

Consideriamo il seguente frammento di codice, in cui vengono dichiarati un array ed una variabile:

```
int a[4];
int x = 5;
```

Nello stato interno del programma, all'indirizzo nella memoria in cui è salvato `a` troviamo l'indirizzo della cella di memoria in cui è salvato il suo primo elemento `a[0]`.

Cosa significa questo? A quale tipo di memoria assomiglia questo schema?

Semplicemente questa memoria è esattamente la stessa che disegneremmo nel caso `a` fosse un puntatore, da cui deduciamo che gli array in `C` sono puntatori, come mostrato in Figura 29.

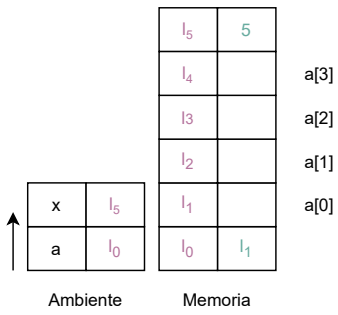


Figura 29. Rappresentazione in memoria di un array. Si noti che gli array in `C` sono puntatori!

Array come input di funzioni. Gli array possono essere passati come parametri di una funzione. Dal momento che non esiste un modo per sapere quanti elementi contiene un array, se vogliamo scrivere una funzione che modifichi o comunque lavori con le celle (o posizioni) di un array, dobbiamo passare in

input oltre all'array stesso anche la sua dimensione. La sintassi per passare un array come parametro di una funzione è mostrata nel seguente Esempio.

Esempio 6.5 (Somma elementi come funzione)**.** Consideriamo di voler scrivere una funzione che, dato un array, ritorni la somma dei suoi elementi, come nell'Esempio 6.4.

```
/*
    Funzione per calcolare la somma degli elementi di un array.
    Argomenti:
        v: array di interi
        dim: numero di elementi in v
    Output:
        int: somma degli elementi di v
*/
int somma(int v[], int dim)
{
    int tot = 0; //var. per memorizzare somma elementi
    int i = 0;
    for(i=0; i<dim; i++) //ciclo sugli elementi dell'array
        tot = tot + v[i]; //incremento tot
    return tot;
}
```

Possiamo anche preparare un funzione che legga in input dall'utente un numero arbitrario di numeri (interi ad esempio), cosi da velocizzare il nostro codice quando testeremo altre funzioni o programmi.

Esempio 6.6 (Lettura di un array come funzione)**.** Consideriamo ora una funzione che prende in input un vettore `v` di dimensione `dim` e memorizza all'interno dello spazio di memoria allocato per `v` dei valori che vengono dati in input dall'utente, ovvero:

```
// Nota: assumo di sapere quanto sia grande l'array (dim)
void lettura(int v[], int dim)
{
    int i = 0;

    /* ciclo sugli elementi dell'array */
    for(i=0; i<dim; i++)
        /* leggo da tastiera l'i-esimo valore */
        scanf("%d", &v[i]);
}
```

Esercizio 6.1. Si disegni la memoria degli ultimi due esempi di questa sezione. Si ponga particolare attenzione agli arrayed al passaggio dei parametri, ed a come funziona la memoria.

Essendo puntatori, posso anche definire funzioni con argomenti o valori di ritorno che sono puntatori (e.g., `int *`, `float *`, etc.). Più avanti ne mostre-

remo qualche esempio.

Array ed aritmetica dei puntatori. Come abbiamo già visto, gli array sono implicitamente dei puntatori all'indirizzo di memoria in cui è contenuto il loro primo elemento, quindi possiamo assegnare un array ad un puntatore:

```
int a[10];
int* p;
/* assegno l'array al puntatore */
p = a;
```

Consideriamo una variabile `p` di tipo puntatore ad intero, ovvero `int* p`. Quando calcoliamo `p + 1` ci spostiamo nella locazione successiva a `p`, ovvero se consideriamo il seguente codice:

```
int a[10];
int* p;
int x;

p = &a[3]

// i prossimi due comandi sono equivalenti
x = *(p+1);
x = *(& a[4]); //  a[4]
```

in `x` avremo esattamente `a[4]`, poiché gli array sono memorizzati in celle di memoria contigue e in `p` abbiamo memorizzato l'indirizzo di `a[3]`.

In generale, se consideriamo un puntatore ad intero `p`, l'espressione `p+i` valuta ad un puntatore che punta `i` locazioni dopo la locazione puntata da `p` (ovvero i interi dopo l'intero puntato da `p`).

> **Nota bene.** (Incrementi interi ed indirizzi) Questa aritmetica funziona perché gli indirizzi di memoria aumentano (di una unità) proprio come se fossero interi.

Usiamo l'aritmetica dei puntatori in un esempio concredo.

> **Esempio 6.7** (Scambiare valori su array)**.** Definiamo una funzione che dato un array `v` di dimensione `dim` e due indici `i, j`, scambia il valore in `v[i]` con quello in `v[j]`.
>
> ```
> void scambia(int v[], int dim, int i, int j)
> {
>     /* sanity check sulla possibilità di effetture lo scambio */
>     if(i<dim && j<dim && i>0 && j>0)
>     {
>         int tmp = v[i];
>         v[i] = v[j];
>         v[j] = tmp;
> ```

```
    }
}
```

Sfruttando esplicitamente l'aritmetica dei puntatori ed il fatto che il nome di un array è in realtà il nome del puntatore alla prima cella in cui è memorizzato l'array, la stessa procedura può esser scritta come:

```
void scambia(int* v, int dim, int i, int j)
{
    /* sanity check sulla possibilità di effetture lo scambio */
    if(i<dim && j<dim && i>0 && j>0)
    {
        int tmp = *(v + i); //v[i]
        *(v + i) = *(v + j); //v[i] = v[j]
        *(v + j) = tmp; //v[j] = tmp
    }
}
```

6.3 Problem solving su array

Ricerca con flag. Consideriamo un semplice problema di ricerca: vogliamo trovare un elemento x all'interno di un array, in pratica restituendo un valore (ad esempio 1) se x fa parte dell'array, o 0 altrimenti.

Esercizio 6.2 (Quanto costa?)**.** Si ragioni su quante operazioni debbano essere effettuate per rispondere a questa domanda - si pensi in modo astratto, senza scrivere codice. Ad esempio, si conti il numero dei confronti.

Possiamo scrivere in logica proposizionale, usando i quantificatori esistenziali ($\exists$) o universali ($\forall$), il predicato associato al nostro calcolo

$$\exists i \in [0, dim).\, a[i] == x$$

dove stiamo assumendo che:

- a sia il nostro array;
- dim sia la dimensione di a.

Nota bene. (Logica proposizionale) Il simbolo $\exists x.p(x)$ significa "esiste un x tale per cui vale il predicato $p(x)$" - qui "esiste un valore i compreso tra 0 e dim (stretto, $0 \leq i < dim$) tale per cui l'i-esimo elemento di a è esattamente x. In modo simile possiamo scrivere $\forall x.p(x)$ significa "per ciascun valore di x vale il predicato $p(x)$".

Esempio 6.8 (Ricerca di base)**.** Ecco una funzione che dato `a` di dimensione `dim` ed `x`, verifica se `x` è un elemento di `a`.

```
int member(int a[], int dim, int x)
{
    // scandisco tutto a
    for(int i = 0; i < dim; i++)
    {
```

```
        // ma non appena trovo x, ritorno 1
        if(a[i] == x)
            return 1;
    }

    // se arrivo qui, x non c'è, ritorno 0
    return 0;
}
```

Esercizio 6.3. Si consideri questa implementazione alternativa, e si confronti con quella dell'Esempio 6.8. Si discuta delle differenze e si creino esempi di input in cui le due funzioni danno risultati analoghi e differenti.

```
int member(int a[], int dim, int x)
{
    for(int i = 0; i < dim; i++)
    {
        if(a[i] == x)
            return 1;
        else
            return 0;
    }
}
```

La funzione dell'Esempio 6.8 è una prima approssimazione a quello che vorremmo davvero considerare come una buona implementazione. Fondamentalmente la funzione non è sbagliata, è solo particolarmente "brutta"[6], secondo un criterio in cui non vorremmo dover avere dei `return` che interrompono il flusso di esecuzione di un ciclo che - per definizione di ciclo `for` - dovrebbe scandire esattamente tutto il vettore `a`.

Cerchiamo di discutere diverse soluzioni alternative. La prima cosa da realizzare è che la scansione in una domanda del genere denota una ricerca "incerta" e quindi di lunghezza indeterminata. Pertanto, sembra più sensato implementarla con un `while` (piuttosto che con un `for`).

Esempio 6.9 (Ricerca con while)**.** Possiamo iterare fino a che non troviamo `x`, e gestire il caso in cui non lo si trovi mai.

```
int member(int a[], int dim, int x)
{
    int i = 0;

    // scandisco tutto a, al massimo
    while(i < dim && a[i] != x) i++;

    // il valore di i mi dice tutto
    if(i == dim) return 1;
```

[6] Qua la questione diventa un poco filosofica - chi analizza un programma del genere potrebbe non rendersi conto del fatto che l'iterazione non durerà esattamente `dim` passi, ma potenzialmente meno. In questo esempio triviale, probabilmente andrebbe bene. In condizioni di codice piu complesso, potrebbe essere difficile comprendere il flusso di esecuzione di un programma del genere.

```
    else return 0;
}
```

Esercizio 6.4. Nell'Esempio 6.10 posso invertire l'ordine nella guardia e testare `(a[i] != x && i < dim)`?

Siamo chiaramente davanti ad una soluzione corretta, ma ancora migliorabile. In pratica, potrei rendere il codice ancora più leggibile (quasi come se volessi leggerlo in linguaggio naturale). Per fare questo, devo utilizzare una variabile ausiliaria, che si chiamerà *flag*.

Questa variabile verrà associata allo "stato" della computazione, inteso come al fatto che si sia o meno trovato l'elemento in questione.

Esempio 6.10 (Ricerca con flag e while)**.** Possiamo iterare fino a che non troviamo `x`, stavolta col flag.

```
int member(int a[], int dim, int x)
{
    int i = 0;
    int trovato = 0; // flag (0: non trovato)

    // scandisco tutto a, al massimo
    while(i < dim && !trovato)
    {
        // cambio status del flag
        if(a[i] === x) trovato = 1;
        i++;
    }

    // il valore di trovato mi dice tutto
    if(trovato) return 1;
    else return 0;
}
```

Ovviamente ci sta scappando di vista il fatto che `trovato` contenga il valore effettivo da restituire.

```
int member(int a[], int dim, int x)
{
    int i = 0;
    int trovato = 0; // flag (0: non trovato)

    // come sopra
    while(i < dim && !trovato)
    {
        if(a[i] == x) trovato = 1;
        i++;
    }

    return trovato;
```

```
}
```

Prima di concludere la ricerca con flag, ne riportiamo l'implementazione usando l'aritmetica dei puntatori.

```
int member(int * a, int dim, int x)
{
    int trovato = 0; // flag (0: non trovato)

    // non come sopra
    while(dim >= 0 && !trovato)
    {
        if(*(a + i) == x) trovato = 1;
        dim--;
    }

    return trovato;
}
```

Nota bene (Avanti o indietro?)**.** Nell'ultimo esempio facciamo anche un uso diverso della variabile `dim`. Cosí facendo non abbiamo nemmeno bisogno della variabile `i`. Questa soluzione è sicuramente quella più leggibile. Quale proprietà dello stack frame utilizza questa funzione per essere "corretta"?

Esercizio 6.5. Scrivere delle funzioni in `C` che calcolino i seguenti predicati logici:

- $\exists i \in [0, dim).a[i] < 0$ (l'array ha almeno un elemento negativo);
- $\exists i \in [0, dim).a[i]\%2 == 0$ (l'array ha almeno un elemento pari);
- $\exists i \in [0, dim - 1).a[i] == a[i + 1]$ (due elementi consecutivi dell'array sono uguali). Si noti come qui l'estremo destro dell'intervallo sia `dim - 1`, e non `dim`, altrimenti si uscirebbe dall'insieme degli indici ammissibili per l'array quando si controlla `a[i+1]`.

Per tutti questi problemi, lo schema di iterazione è lo stesso ottenuto modificano l'Esempio 6.8, ovvero si ispezione l'array attraverso un ciclo `while` che tenga conto sia della dimensione dell'array che di una variabile flag.

Proprietà globali di un array. Come abbiamo mostrato un esempio di ricerca con associato un predicato esistenziale, possiamo trovarne di alternativi con predicato universale. Questo ci permette di guardare le proprietà globali di un array, ovvero quelle che riguardano proprietà di tutta la struttura.

Esempio 6.11 (Quantificatori universali)**.** I seguenti predicati logici riguardano proprietà globali di un array:

- $\forall i \in [0, dim).a[i] < 0$ (array con elementi tutti negativi);
- $\forall i \in [0, dim).a[i]\%2 == 0$ (array con elementi tutti pari);
- $\forall i \in [0, dim - 1).a[i] == a[i + 1]$ (array con elementi tutti uguali).

Ragioniamo di come scrivere in C una funzione che calcoli il secondo predicato della lista, ovvero quello che chiede di verificare se tutti gli elementi dell'array sono pari. Inizialmente, potremmo pensare a questa soluzione.

```
int tuttipari(int a[], int dim)
{
    int pari = 0; //var. da ritornare (flag)

    /* ciclo su tutti gli elementi dell'array */
    for(int i=0; i<dim; i++)
    {
        /* i-esimo elemento è pari/dispari */
        if(a[i]%2==0)
            pari = 1;
        else
            pari = 0;
    }

    return pari;
}
```

Esercizio 6.6. Si consideri l'implementazione qui sopra, si mostri si è corretta o meno, e si dia un esempio di input per il quale il comportamento osservato non corrisponde con quello atteso dal predicato.

Chiaramente questa soluzione è sbagliata, perché non si porta dietro il risultato del calcolo collettivo del predicato. Questa implementazione è invece corretta, e si ottiene come minima modifica della precedente.

```
int tuttipari(int a[], int dim)
{
    int pari = 1; //var. da ritornare (flag)
    /* ciclo su tutti gli elementi dell'array */
    for(int i=0; i<dim; i++)
    {
        /* i-esimo elemento è pari */
        if(a[i]%2==0)
            pari = pari * 1;
        /* i-esimo elemento è dispari */
        else
            pari = pari * 0;
    }
    return pari;
}
```

Esercizio 6.7. Si argomenti del perché questa implementazione sia invece corretta, e si mostri sugli esempi definiti nell'esercizio precedente il risultato di questa funzione.

Proviamo a sviluppare comunque un ulteriore ragionamento sulla proposizio-

ne, chiamata $P_\forall$ per semplicità, ovvero

$$P_\forall \equiv \forall i \in [0, dim).a[i]\%2 == 0\,.$$

A parole, sono tutti pari se e solo se nessuno è dispari. Quindi

$$P_\forall \iff \nexists i \in [0, dim).a[i]\%2 == 1 \equiv P_\nexists$$

ovvero $P_\nexists$ definisce il predicato che calcola "non esiste un indice di un numero dispari" (indi per cui sono tutti pari). Si noti che $P_\nexists$ è la *negazione* del predicato di ricerca semplice per un numero dispari

$$\exists i \in [0, dim).a[i]\%2 == 1\,.$$

Quest'ultimo risultato lo sappiamo calcolare perché trattasi esattamente della ricerca con flag, alla quale complementiamo (i.e., invertiamo) il risultato di finale (la variabile `trovato`).

```c
int tuttipari(int a[], int dim)
{
    int i = 0;
    int trovato = 0; // flag: esiste un dispari

    // pattern di ricerca
    while(i < dim && !trovato)
    {
        // predicato specifico - è dispari?
        if(a[i] % 2 == 1) trovato = 1;
        i++;
    }

    // negazione (o complemento) del risultato
    return !trovato;
}
```

Questa funzione può anche essere invertita per eliminare le negazioni, in tal caso partendo da una ipotesi diversa (opposta).

```c
int tuttipari(int a[], int dim)
{
    int i = 0;
    int ok = 1;

    // pattern di ricerca
    while(i < dim && ok)
    {
        // predicato specifico - è dispari?
        if(a[i] % 2 == 1) ok = 0;
        i++;
    }
```

```
    // risultato senza negazione
    return ok;
}
```

Esercizio 6.8. Si implementino i predicati dell'Esempio 6.11.

> **Nota bene.** Si noti che siamo partiti alludendo il fatto che, per proprietà universali, avremmo dovuto guardare tutti gli elementi del vettore. Poi, attraverso una riduzione logica, siamo riusciti a scrivere il programma con un quantificatore esistenziale. A questo punto, siamo proceduti con l'implementazione basata su `while`, dato il carattere incerto della durata dell'iterazione.

Problemi di conteggio ed aggregazione. Un'altra tipologia di problemi su array sono i problemi di conteggio, ad esempio contare quanti elementi pari ci sono in un array, oppure quanti positivi.

Esercizio 6.9. Si definisca input/output di una funzione che conta quanti elementi pari ci sono in un array. Si crei un esempio; quanto costa determinare la risposta a questa domanda, in termini di operazioni da fare sull'input?

Per risolvere questo tipo di problema dobbiamo introdurre una variabile di tipo *contatore* (i.e., un intero che conti quello che dobbiamo riportare).

> **Esempio 6.12** (Conteggio dei pari)**.** Per risolvere questo problema possiamo definire, dato un array di interi `a` di dimensione `dim`, l'insieme
>
> $$I = \{i | i \in [0, dim) \wedge a[i]\%2 == 0\}$$
>
> (ovvero l'insieme degli indici che si riferiscono a numeri pari all'interno dell'array `a`) e calcolare la sua cardinalità, denotata $|I|$.
>
> ```
> int contapari(int a[], int dim)
> {
>     // inizialmente ne ho visti 0 pari
>     int contatore = 0;
>
>     for(int i=0; i<dim; i++)
>     {
>         /* se l'i-esimo elemento di a è pari */
>         if(a[i]%2==0)
>             /* incremento il contatore */
>             contatore++;
>     }
>
>     // restituisco il contatore
>     return contatore;
> }
> ```

La forma generale di queste funzioni richiede quindi di calcolare la cardinalità di un insieme definito spesso tramite un predicato $p(a, i)$ che dipende

dall'array e dall'indice, ad esempio

$$|\{i|i \in [0, dim) \wedge p(a, i)\}|\,.$$

Per diretta simiglianza con i problemi di conteggio, affrontiamo quelli di aggregazione. In questi problemi finiamo spesso a definire una variabile di tipo*accumulatore* che serve per calcolare una proprietà sugli elementi associati al predicato $p(a, i)$. Ad esempio, una somma totale

$$I_{p(a,i)} = \sum_{x \in I} x$$

dove $I = \{i|i \in [0, dim) \wedge p(a, i)\}$.

Esempio 6.13 (Aggregazione per la somma dei pari)**.** Dato un array di interi `a` di dimensione `dim`, si vuole calcolare la somma di tutti gli elementi pari di `a`.

```
int sommapari(int a[], int dim)
{
    int somma = 0; // variabile accumulatore

    for(int i=0; i<dim; i++)
    {
        /* se l'i-esimo elemento è pari */
        if(a[i]%2 == 0)
            /* lo aggiungo alla variabile accumulatore */
            somma += a[i];
    }

    // resitutisco l'accumulatore
    return somma;
}
```

Minimo di un array. Questo tipo di calcolo riunisce tutte le intuizioni viste fino ad ora. Per risolverlo combineremo, almeno inizialmente, anche più funzioni assieme.

Esempio 6.14 (Elemento minimo)**.** Dato un array `a` di dimensione `dim`, si vuole ricercare l'elemento minimo di `a`, ovvero il più piccolo.

Scritto come predicato logico, questo problema è:

$$\exists i \in [0, dim).(a[i] == x \wedge \forall j \in [0, dim).a[j] \geq x)\,.$$

Come primo passo, partiamo da una funzione che calcola il predicato

$$\forall j \in [0, dim).x \leq a[j]$$

ovvero verifica se x è più piccolo di ogni elemento dell'array `a`. Questo è

equivalente a

$$\exists j \in [0, dim).x > a[j]\,.$$

Esempio 6.15 (Minore di tutti)**.** Questo problema può esser risolto con la seguente funzione:

```
int minoreditutti(int a[], int dim, int x)
{
    int i = 0;
    int ok = 1; //flag, "nessun elemento minore di x"

    // pattern con flag!
    while(i<dim && ok)
    {
        /* se l'i-esimo elemento è minore di x */
        if(x>a[i])
            /* modifico flag */
            ok = 0;
        i++;
    }

    // come negli esempi di prima
    return ok;
}
```

Adesso il problema originale di trovare il minimo elemento nell'array `a` diventa:

$$\exists i \in [0, dim).(a[i]) == x \wedge \texttt{minoreditutti}(a, dim, a[i])\,,$$

e possiamo usare `minoreditutti`.

Esempio 6.16 (Ricerca del minimo)**.**

```
int minimo(int a[], int dim)
{
    int i = 0;
    int min; //variabile da ritornare
    int trovato; //flag, "ho trovato il minimo"

    while(i<dim && !trovato)
    {
        /* se l'elemento i-esimo è più piccolo di tutti gli altri  */
        if(minoreditutti(a, dim, a[i]))
        {
            /* ho trovato il minimo e modifico il flag */
            trovato = 1;
            min = a[i];
        }

        i++;
    }

    // Si ricorda il minimo
    return min;
```

```
}
```

Definendo la funzione ausiliaria `minoreditutti` abbiamo ricondotto il problema originale ad un problema di ricerca con flag.

Esercizio 6.10 (Quanto costa?)**.** Si conti il numero di operazioni fatte da queste due funzioni, assumendo l'array di dimensione $n > 0$.

Questo non è il modo migliore di risolvere questo problema, poiché richiede di confrontare ciascun elemento con tutti gli altri, ovvero questo algoritmo è *quadratico* nella dimensione dell'array.

Un modo per rendere più efficiente questa procedura è:

- supporre che il primo elemento dell'array sia il minimo (e lo si assegna a `min`);
- confrontarlo con tutti gli i valori seguenti;
- appena un valore (supponiamo `a[j]`) è minore del primo elemento lo memorizziamo in `min`;
- si ripete la procedura confrontando il valore in `min` (ovvero `a[j]`) con tutti i valori seguenti (ovvero gli elementi a partire da `a[j+1]`) e così via, fino a che non si è finito di ispezionare l'array.

Questa procedura è lineare nella dimensione dell'array in input.

Esercizio 6.11. Si definisca la funzione qui descritta a parole, oltre alla precedente. Si utilizzino delle `printf` stampando un carattere (e.g. `'*'`), laddove c'e' una operazione effettuata dall'algoritmo implementato. Usando un vettore di 10 elementi, si confronti il numero di asterischi stampati dalle due funzioni.

6.4 Stringhe

In `C` le **stringhe** sono vettori di `char`. La loro struttura è particolare in quanto, per costruzione, l'ultimo elemento di una stringa è un carattere di terminazione, rappresentato da \0, come mostrato in Figura 30.

Per questo motivo, se la parola che vogliamo memorizzare nella stringa ha n caratteri, dobbiamo dichiarare un array di lunghezza $n + 1$.

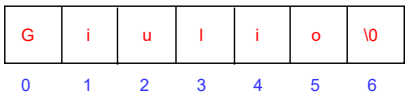


Figura 30. Rappresentazione di una stringa: per scrivere la parola 'Giulio', che ha 6 lettere, occorre allocare un array di 7 elementi, poiché va lasciato spazio per l'elemento di terminazione.

L'identificatore di formato utilizzato dalle funzioni `scanf, printf` è `%s`.

Nota bene. Dal momento che le stringhe sono array, e abbiamo visto che in memoria gli array sono dei puntatori al loro primo elemento, per ricevere una stringa in input la funzione `scanf` non ha bisogno dell'operatore & (il nome della stringa è già di per sé un indirizzo!). Ovvero per leggere una stringa: `scanf("%s", nome_stringa)`.

Le seguenti sono tutte possibili dichiarazioni con inizializzazione di stringhe in C:

```
char stringa1[6] = {'P', 'i', 'p', 'p', 'o', '\0'};
//il carattere di terminazione è inserito dal compilatore in questo caso
char stringa2[6] = "Pippo";
```

Per prendere in input da tastiera una stringa invece esistono le seguenti possibilità:

```
//sfruttando esplicitamente il fatto che è un array
char stringa1[6];
for(int i=0; i<6; i++)
    //leggo elemento ad elemento
    scanf("%c", &s[i]); //qui ci va & poiché è il singolo elemento
stringa1[5] = '\0';
//utilizzando %s
char stringa2[6];
//legge una stringa fino a che non viene trovato un carattere di spaziatura
//poi al posto del carattere di spaziatura viene automaticamente inserito '\0'
//che quindi non va inserito manualmente
scanf("%s", stringa2);
```

Per manipolare una stringa è invece necessario accedere ai singoli caratteri, come si fa tipicamente con gli array, ricordandosi del fatto che l'ultimo indice ammissibile corrisponde al carattere di terminazione.

Esempio 6.17. Scrivere un programma che, data una stringa in input, restituisca 1 se il numero di vocali presenti nella stringa è dispari, 0 altrimenti. Una possibile implementazione è la seguente:

```
int vocali_dispari(char s[])
{
    int i = 0; //var. iterazione
    int num_voc = 0; //contatore per il numero di vocali
    while(s[i]!=0) //scorro tutta la stringa
    {
        //controllo se l'i-esimo carattere è una vocale
        if(s[i] == 'a' || s[i] == 'o' || s[i] == 'i' || s[i] == 'o' || s[i] == 'u')
            num_voc += 1;
        i++;
    }
    if(num_voc%2==0) //numero pari di vocali
        return 1;
    else //numero dispari di vocali
        return 0;
```

```
}
```

Esempio 6.18 (Calcolare la lunghezza di una stringa)**.** Il seguente programma calcola iterativamente la lunghezza di una stringa in C:

```c
int lunghezza(char s[])
{
    int l = 0;
    //sfruttiamo il carattere di terminazione
    while(s[l]!='\0')
        l++;
    return l;
}
```

Esercizio 6.12. Implementare una funzione ricorsiva che calcoli la lunghezza di una stringa.

7 Memoria dinamica

La memoria "statica", ovvero quella che abbiamo visto fino ad ora, composta di frame corrispondenti a blocchi contenuti in un programma, ha una limitazione fondamentale. In pratica, sullo stack (coppia ambiente/ memoria), si possono allocare solamente array di dimensione pre-determinata . Quindi come possiamo scrivere un programma che prenda in input dei numeri fino a che non trova un valore negativo (senza sapere a priori quanti saranno), e li memorizzi? Oppure un programma che legge da un file una serie temporale di valori, senza sapere quanti questi saranno?

Si ricordi la dichiarazione `int a[10];` per indicare un array di 10 elementi.

Esercizio 7.1. Si pensi - ad alto livello - ad almeno 3 situazioni distinte in cui potremmo non conoscere la dimensione del dato con cui lavoreremo.

Per gestire questo tipo di programmi intrinsecamente dinamici abbiamo bisogno appunto di una memoria "dinamica", che chiameremo **heap** (a volte denotata ζ).Lo heap è una parte diversa della memoria in cui le variabili non vengono allocate dai blocchi, ma tramite delle operazioni fatte esplicitamente dal programma.

In particolare, lo heap è una tabella di associazione

$$\langle \text{locazione}, \text{valore} \rangle$$

che non ha i frames; gli indirizzi dello heap vengono denotati con h_0, h_1, ... per distinguerli dagli indirizzi dello stack. L'assenza di frames nello heap ovviamente permette di avere variabili che vivono più a lungo del blocco in cui sono definite, contrariamente alla memoria statica che fornisce uno stack.

Il nostro nuovo modello di stato interno del programma è quindi mostrato in Figura 31, come naturale estensione della coppia ambiente/ memoria.

7.1 Allocazione della memoria

Non essendoci un meccanismo implicito di gestione dello heap - come quello che ci fornisce lo stack - dobbiamo gestire la memoria dinamica *esplicitamente*. Per farlo usiamo funzioni della libreria standard `stdlib`, inclusa nei programmi con `include`

```
#include <stdlib.h>
```

La memoria dinamica si alloca mediante diverse tipi di funzione - tutte che appartengono ad una unica grande famiglia - ma in questa dispensa ci occupiamo solamente di `malloc`. Questa, come altre funzioni, creano una nuova variabile nello heap e restituisce un puntatore alla variabile. Con riferimento a Figura 31, ad esempio, alloca lo spazio necessario e ritorna h_0.

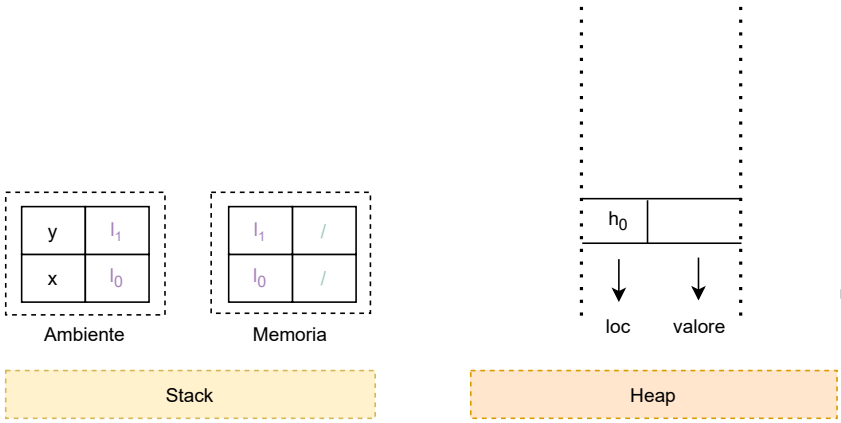


Figura 31. Stato interno di un programma avente variabili sia nello stack (coppia ambiente/ memoria) che nello heap (alla locazione h_0); la prima si considera memoria "statica", la seconda "dinamica". Lo heap, sempre accessibile tramite un sistema ad indirizzi (i.e., h_0), a differenza dello stack non è caratterizzato da una struttura a frames.

`malloc` richiede in input la quantità di memoria richiesta, che dipende dal tipo della variabile che creiamo e può esser calcolata tramite la funzione `sizeof(nome_tipo)`. La sintassi completa per una allocazione è:

```c
#include<stdlib.h> //per poter usare malloc

void main()
{
    int* a = (int *)malloc(sizeof(int)); //alloco spazio per 1 intero usando sizeof(int)
}
```

Esercizio 7.2. Si disegni la memoria di questo programma.

Nota bene (Variabili senza nome)**.** Il meccanismo di allocazione dinamica del `C` ci permette di creare delle variabili che non hanno un nome - inteso come un nome nell'ambiente - ma solamente un indirizzo associato.

Nota bene (Cast esplicito)**.** Si noti il cast esplicito al puntatore restituito da `malloc`. Il cast non è necessario, ma serve per chiarire il tipo del puntatore e per rendere codice `C` compilabile in `C++`. Il motivo per cui si esegue è perché il puntatore ritornato da `malloc` è di tipo `void *`

```c
void* malloc( size_t size );
```

ovvero del tipo più generale di tutti (`void`). Questi due codici sono dunque equivalenti

```c
int *ptr, *ptr2;
ptr = malloc(sizeof(int)); /* senza cast */
ptr2 = (int *)malloc(10 * sizeof(int)); /* con cast */
```

e nel corso degli esempi a volte useremo il cast, a volte no.

Esempio 7.1 (Allocazione di un intero)**.** Si considerino i seguenti modi di dichiarare ed inizializzare un intero.

```c
{
    int x = 10; //var. nello stack
```

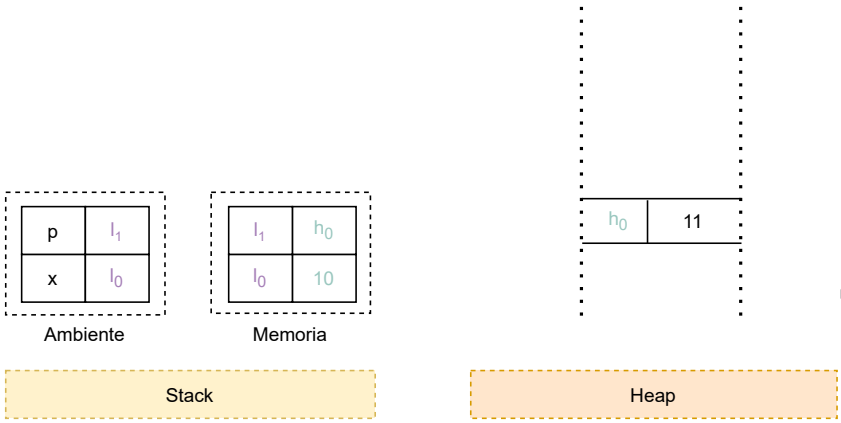


Figura 32. Stato interno del programma dell'Esempio 7.1

```
    int* p;
    p = malloc(sizeof(int)); //var. nello heap
    *p = x + 1; //*p = 11
}
```

Lo stato interno di questo programma è mostrato in Figura 32. Quando si crea una variabile nello heap viene creato nello stack un puntatore contenente l'indirizzo nello heap in cui tale variabile è memorizzata.

Ovviamente possiamo allocare anche memoria per un array. Cosa distingue un array di interi da un intero? In realtà se ricordiamo la memoria di un array (Figura 29) le posizioni sono tutte allocate in modo contiguo, quindi un array di n elementi occupa esattamente memoria per n interi, da cui possiamo usare una espressione

```
10 * sizeof(int)
```

per ottenere un puntatore - quindi un array - concettualmente simile a `int a[10]`, ma allocato sullo heap invece che sullo stack.

Esempio 7.2 (Allocazione di un array di interi)**.** Si consideri finalmente un programma che alloca correttamente memoria per una array la cui dimensione non sarà mai nota se non a tempo di esecuzione.

```
{
    int x; // var. nello stack

    scanf("%d", &x); // input dinamico

    p = malloc(x * sizeof(int)); //array di x elementi, nello heap

    ... // codice che lavora con p array
}
```

Lo stato interno di questo programma è mostrato in Figura 33. Notare la somiglianza con la rappresentazione in stack, ovviamente.

Nota bene (Heap e scoping)**.** La variazione dello heap non viene influenzata dai blocchi, una eccezione alla regola di scoping classica.

```
{
```

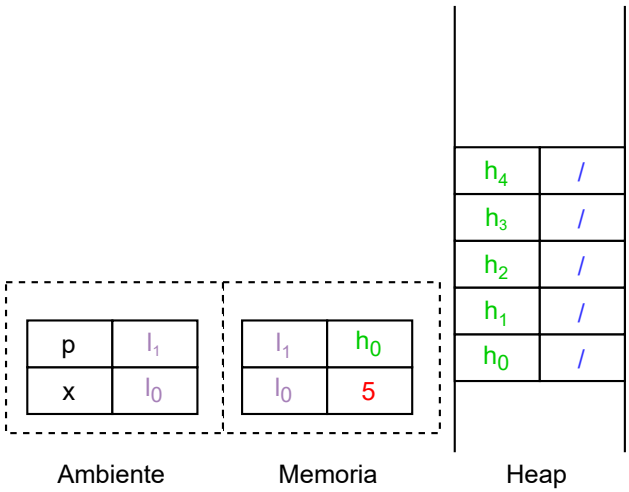


Figura 33. Stato interno del programma dell'Esempio 7.2, ovvero un array allocato dinamicamente nello heap.

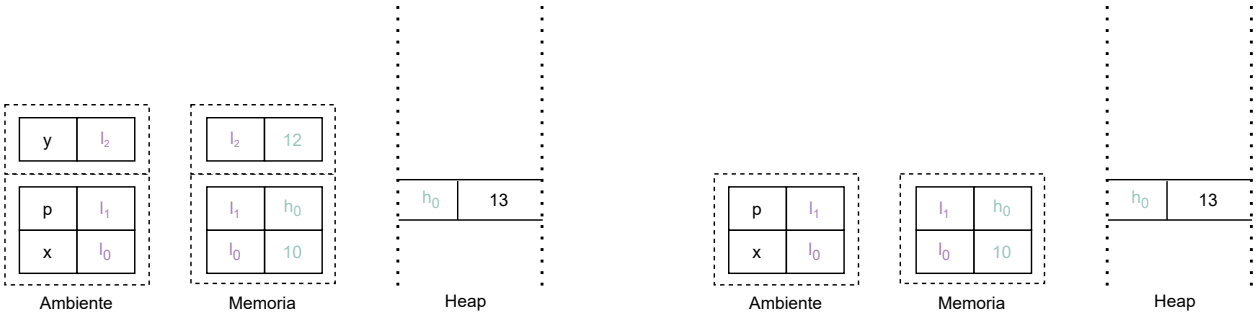


Figura 34. La variazione dello heap non viene influenzata dai blocchi. Lo stato del programma mostra `p` valido anche dopo la chiusura del blocco.

```
    int x = 10;
    int* p;

    // blocco
    {
        int y = 12;

        // heap
        p = malloc(sizeof(int));
        *p = y + 1;
    }

    // p è valido
    x = *p;
}
```

Lo stato interno di questo programma è rappresentato in Figura 34, dove si può apprezzare come le variabili memorizzate sullo heap sopravvivano alla fine del blocco in cui sono state definite.

La gestione della memoria dinamica deve essere esplicita, quindi in caso di gestione inappropriata si creano dei cosiddetti "leaks" di memoria.

Esempio 7.3 (Memory leak). Allocazioni successive e conseguenti ri-assegnamenti fanno perdere il reference della zona di memoria, ma la memoria rimane allocata.

Si consideri il seguente programma:

```
{
    int x = 0;

    // primo p
    int* p = malloc(sizeof(int));
    *p = 12;

    // secondo p (il primo reference viene perso)
    p = malloc(sizeof(int));
}
```

Lo stato interno di questo programma è schematizzato in Figura 34. Ovvero se si dà lo stesso nome a due celle con indirizzo diverso sullo heap, quella che è stata dichiarata prima rimane *orfana*, i.e. non è più accessibile da nessun puntatore nel programma.

La memoria dinamica che rimane allocata ma il cui indirizzo non è più accessibile (perchè nessuna variabile lo contiene), si chiama **garbage**. In alcuni linguaggi di programmazione - ad esempio in `Java` - esiste il garbage collector, uno strumento responsabile di ispezionare l'heap e rendere ri-utilizzabili le zone di memoria diventate garbage; in `C` questo strumento non esiste.

Un altro semplice esempio di memory leak, stavolta generato dalle funzioni, potrebbe essere il seguente.

```
#include <stdlib.h>

void funzione_che_alloca(void)
{
    /* alloca un array di 45 floats */
    float *a = malloc(sizeof(float) * 45);

    /* codice che usa 'a' */
    ....

    /* return al main (void) */
}

int main(void) {
    funzione_che_alloca();

    /* il puntatore 'a' non esiste più, ma l'array di 45 floats è ancora allocato (memory leak). */
}
```

La memoria dinamica va quindi gestita esplicitamente, i.e. deallocata, attraverso una funzione `free(nome_var)`. La `free` prende in input un puntatore ad una zona di memoria allocata con `malloc`, e la libera.

```c
int *p = malloc(sizeof(int));
*p = 12;

free(p); //libero la memoria
```

> **Nota bene.** Dopo che una zona di memoria è stata liberata tramite `free`, essa non è più accessibile!

Il pattern di programmazione corretto per l'esempio precedente diventa quindi il seguente.

```c
#include <stdlib.h>

void funzione_che_alloca(void)
{
    /* alloca un array di 45 floats */
    float *a = malloc(sizeof(float) * 45);

    /* codice che usa 'a' */
    ....

    /* libera la memoria, e  return al main (void) */
    free(a);
}

int main(void) {
    funzione_che_alloca();

    /* il puntatore 'a' non esiste più, e l'array è stato liberato */
}
```

8 Liste linkate

Con gli strumenti presentati fino ad ora non possiamo ancora lavorare, ad esempio, con array di dimensione non conosciuta a priori. Per poterlo fare, la struttura dati più facile che possiamo utilizzare è una **lista linkata**, ovvero una lista di elementi collegati (dello stesso tipo, almeno in `C`) che può avere dimensione variabile (i.e., crescere indefinitivamente).

La differenza tra array e lista linkata è schematicamente rappresentata in Figura 35, dove si intuisce come gli elementi della lista non siano contigui come quelli dell'array. Invece, si noti che gli elementi sono collegati attraverso un puntatore; il termine "linked list" significa proprio che ogni elemento contiene un puntatore al successivo. Il caso speciale dell'ultimo elemento viene gestito da un valore particolare, che ci informa che non ci sono ulteriori elementi nella lista (nelle figure denotato con il simbolo `/` invece che dalla freccia).

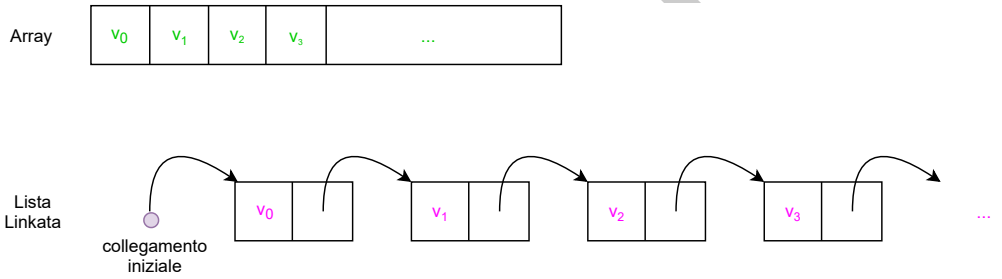


Figura 35. Differenza tra array e lista linkata. Si noti la memoria contigua nel primo caso, ed il link tra elementi della lista nel secondo caso.

> **Nota bene** (Allocazione delle liste)**.** Essendo una struttura dati dinamica, è naturale immaginarla legata ai meccanismi di memoria dinamica discussi precedentemente. Infatti, le liste linkate sono *allocate nello heap*, in celle di memoria non necessariamente contigue.
>
> Inoltre, al contrario degli array che hanno un *accesso diretto*, le liste hanno un *accesso sequenziale*: l'unico punto di accesso ad una lista linkata è il collegamento iniziale, e per ottenere l'elemento i-esimo bisogna scorrere tutti gli elementi che lo precedono.
>
> Per accedere, ad esempio, al quarto elemento di un array `a` è sufficiente fare `a[3]`; nelle liste questo non è possibile ma è necessario fare un algoritmo di camminamento che, a partire dal primo elemento, attraversa la lista fino all'elemento cercato.

Le liste sono strutture dati flessibili e semplici da modificare; nel seguito vedremo come crearle in linguaggio `C`, ed eseguire alcune operazioni fondamentali di modifica (i.e., inserzione e cancellazione di elementi).

Un notevole vantaggio delle liste, rispetto agli array, è che si possono aggiungere (o rimuovere) elementi semplicemente modificando i collegamenti. Sup-

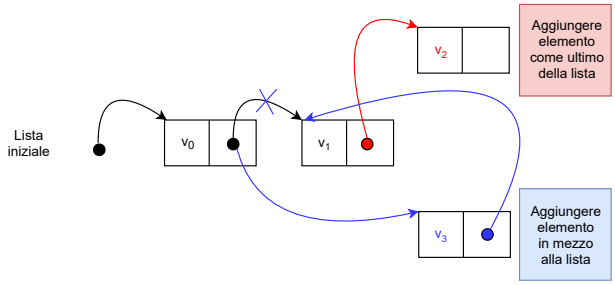


Figura 36. Aggiungere elementi in una lista.

poniamo di dover aggiungere un nuovo elemento d ad una lista, si possono distinguere diversi casi a seconda della posizione in cui vogliamo inserirlo:

- alla fine della lista, ovvero in coda: in tal caso dobbiamo inserire come valore del collegamento dell'attuale ultimo elemento della lista, l'indirizzo del nuovo elemento, e garantire che il nuovo elemento sia il termine della lista (i.e., non punti ad altri elementi);
- in una posizione intermedia della lista: in tal caso dobbiamo cambiare una serie di puntatori. Ad esempio supponiamo di voler aggiungere v_3 tra v_0 e v_1: eliminiamo prima il collegamento tra v_0 e v_1, creiamo un collegamento tra v_0 e v_3 ed infine creiamo un collegamento tra v_3 e v_1. Un ragionamento analogo vale per inserire un elemento in testa alla lista, ovvero come primo elemento: in questo caso v_0 è ciò che in Figura 35 viene chiamato *collegamento iniziale*.

Queste operazioni sono schematicamente mostrate in Figura 36.

> **Nota bene.** Nel caso degli array, ogni volta che si deve aggiungere un elemento occorre distruggere l'array e ricrearne uno di dimensione opportuna con gli elementi richiesti nella posizione giusta.

8.1 Definizione di liste

Per implementare una lista in `C` - supponiamo una lista di interi per semplicità - è necessario quindi utilizzare dei puntatori come collegamenti. Per quel che riguarda la rappresentazione degli elementi della lista, questi si rappresentano con una `struct`, ovvero un *tipo composto*, costruito a partire dai *tipi primitivi* (come `int, float, double, ...`).

La sintassi per definire una struct che descrive un elemento di una lista linkata di interi è la seguente:

```
struct elemento{
    //valore memorizzato
    int info;

    //puntatore al prossimo elemento (dello stesso tipo)
    struct elemento* next;
}
```

La `struct` (structure abbreviato in struct in inglese) rappresenta un gruppo di variabili, anche di tipo diverso, aggregate insieme con un unico nome. In questo le struct assomigliano ai vettori, ma a differenza di questi le variabili non sono ordinate e possono avere anche tipo diverso.

> **Nota bene** (Tipo ricorsivo)**.** Notate che stiamo definendo un tipo "ricorsivo", nel senso che una `struct elemento` dipende a sua volta da un puntatore ad una altra `struct elemento`, ovvero `next`. Il fatto che un tipo sia definito per ricorsione solitamente è una ottima indicazione del fatto che le funzioni che lo manipolano possano essere implementate usando uno schema ricorsivo (e, per altro, anche dello schema stesso).

Si può dare un nome ad una `struct` attraverso il comando `typedef`, il quale crea un tipo "sintattico" che possiamo usare nel nostro programma per semplificare la scrittura dello stesso.

```
typedef struct elemento ElementoDiLista;
```

Nel caso in questione introduciamo un tipo che rappresenta un elemento della lista. Si noti che `typedef` è puramente sintattico, nel senso che il programma a tempo di compilazione, potrebbe sostituire le occorrenze di `typedef` con i tipi concreti (e.g., qui con `struct elemento`) e funzionare perfettamente.

Continuando per la definizione della nostra lista completa, al fine di riferirci alla lista in maniera generale e non ad un suo specifico elemento, possiamo definire il seguente tipo:

```
typedef ElementoDiLista* ListaDiElementi;
```

Quest'ultimo è essenzialmente un puntatore che descrive ciò che abbiamo chiamato *collegamento iniziale*.

Ricapitolando, nel seguito di questa Sezione, faremo riferimento ad una lista linkata come ad una struttura dati definita nel seguente modo:

```
/* dichiarazione */
struct elemento{
    //valore memorizzato
    int info;
    //collegamento al prossimo elemento (stesso tipo)
    struct elemento* next;
}
/* naming */
typedef struct elemento ElementoDiLista;
typedef struct ElementoDiLista* ListaDiElementi;
```

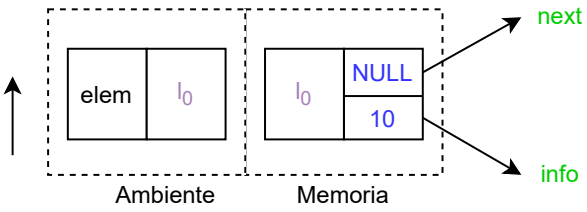


Figura 37. Stato interno del programma durante l'inizializzazione di un elemento di lista nella memoria statica. Da notare che la cella di memoria riservata al valore di `elem` è divisa in sotto-celle contenenti gli attributi della `struct`.

8.2 Creazione di liste

Lista con allocazione statica. Consideriamo il seguente frammento di codice, che alloca staticamente un elemento di una lista linkata:

```
int main(){
    ElementoDiLista elem;
    elem.info = 10;
    elem.next = NULL;
}
```

Per accedere al campo opportuno di una `struct` come `elem` possiamo utilizzare la sintassi

```
nome_struct.nome_campo = valore
```

e quindi ad esempio `elem.info`. La stato interno di questo programma è rappresentato in Figura 37.

> **Nota bene** (Puntatore `NULL`). La parola chiave `NULL` è un modo per rappresentare un puntatore che concettualmente non contiene alcun indirizzo. Si può di fatto assegnare `NULL` ad un puntatore.

Lista con allocazione dinamica. Il seguente frammento di codice mostra come allocare in modo dinamico (ovvero sullo heap) una lista linkata:

```
int main(){
    ListaDiElementi lista;

    //allocazione dinamica
    lista = malloc(sizeof(ElementoDiLista));

    //accesso con dereferenziazione
    (*lista).info = 10; // equivalente a lista->info
    (*lista).next = NULL;
}
```

In questo caso lo stato interno del programma è come in Figura 38. Dal momento che `ListaDiElementi` è un puntatore, per accedere e inizializzare i

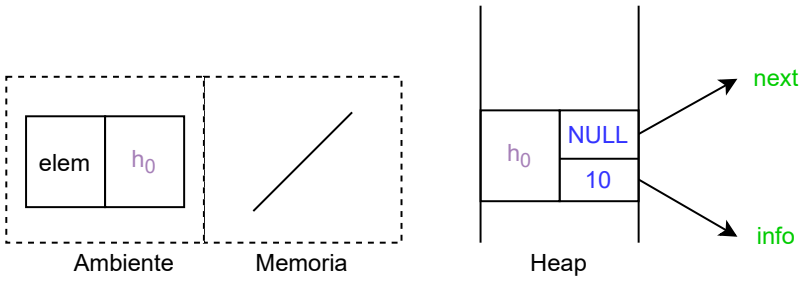


Figura 38. Stato interno del programma nel caso di allocazione dinamica di una lista linkata. La rappresentazione è analoga al caso precedente, ma questa volta la lista vive nello heap.

suoi campi occore usare l'operatore di dereferenziazione.

Nota bene (Sintassi alternativa)**.** Per accedere agli attributi di una `struct` si distinguono due sintassi:

- se la `struct` è data per valore (come la nostra variabile `ElementoDiLista` nell'esempio precedente), allora si usa la sintassi `nome_struct.nome_attributo`;
- se la `struct` è data per indirizzo (ovvero stiamo lavorando con un puntatore alla `struct`, nel nostro caso `ListaDiElementi`), si hanno due possibilità: o si usa la dereferenziazione, ovvero `(*nome_puntatore).nome_attributo`, oppure si usa l'operatore `->`, ovvero `nome_puntatore->nome_attributo` (senza dereferenziazione esplicita).

Esempio 8.1 (Lista con i primi N numeri)**.** [Lista di interi] Si vuole creare una lista di dimensione N contenente gli elementi $1, \ldots, N$. Abbiamo tutti gli strumenti per poterlo fare, se mettiamo assieme i principi di memoria dinamiche, liste linkate, e cicli iterativi.

```
//definizione tipo
struct elemento{ int info; struct elemento* next;}
typedef struct elemento ElementoDiLista;
typedef struct ElementoDiLista* ListaDiElementi;

int main(){
    //creo una lista vuota
    ListaDiElementi lista = NULL;

    //numero elementi lista
    N = 10;

    //creo il primo elemento della lista
    ListaDiElementi new = malloc(sizeof(ElementoDiLista));
    (*new).info = 1;
```

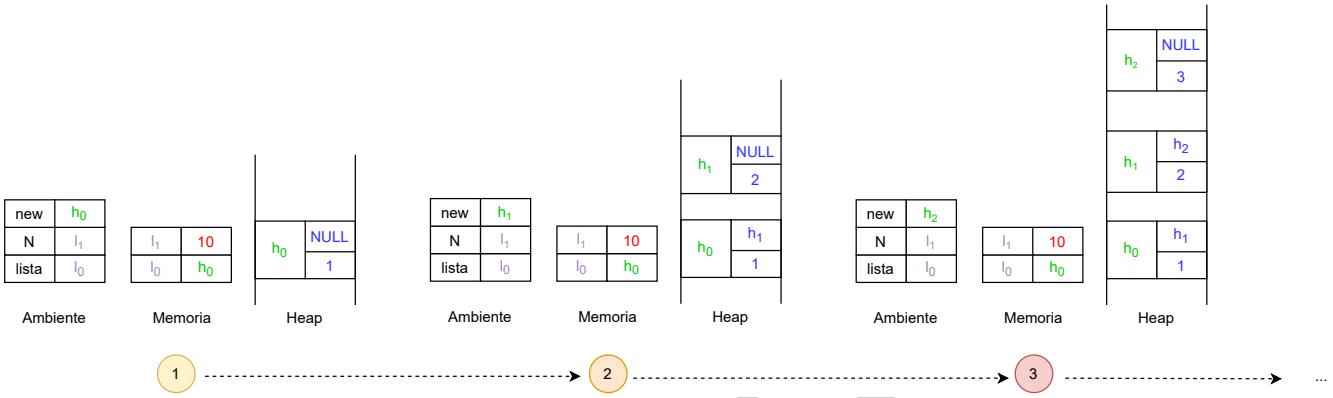


Figura 39. Stato interno del programma dell'Esempio 8.1, mostrando solamente i passi per l'inserzione dei primi 3 elementi nella lista.

```
    //collego il primo elemento al collegamento iniziale della lista
    lista = new

    //aggiungo elementi alla lista
    for(int i=2; i<N; i++)
    {
        //prossimo elemento, allocato e linkato
        (*new).next = malloc(sizeof(ElementoDiLista));

        //sposto puntatore in testa
        new = new->next;

        //assegno nuovo valore
        new.info = i;
    }

    // chiudo la lista, assicurando che non ci sia next non nullo
    new->next = NULL;
}
```

Lo stato interno di questo programma è rappresentato in Figura 39.

Esercizio 8.1. Scrivere un programma che prenda in input da un utente numeri interi fino a che un valore negativo non viene inserito. Il programma deve creare due liste:

- ogni numero pari preso in input lo si memorizza in un lista chiamata `interi_pari`;
- i numeri dispari sono invece inseriti in un'altra lista chiamata `interi_dispari`.

Alla fine gli elementi della lista più grande tra le due vengono stampati, insieme alla somma totale dei valori in ciascuna lista. **Nota:** si richiede che la lunghezza della lista sia calcolata da una funzione che prende in input una lista, e restituisce un intero. Lo stesso per il calcolo della somma totale dei valori inseriti in una lista.

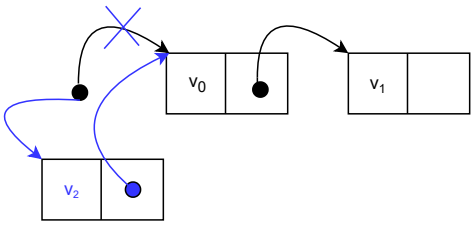


Figura 40. Rappresentazione schematica dell'aggiunta di un elemento in testa ad una lista.

8.3 Manipolazione di liste

Vogliamo definire un algoritmo generale che possa aggiungere un elemento ad una lista; cosi facendo potremmo creare liste di lunghezza arbitraria.

Inserire elemento in testa alla lista. Data una lista linkata, vogliamo inserire un nuovo elemento in testa alla lista. Per farlo occorre eliminare il collegamento tra *collegamento iniziale* e il primo elemento e creare due nuovi collegamenti: uno tra il punto iniziale e il nuovo elemento, un altro tra il nuovo elemento e la vecchia testa della lista, come qualitativamente mostrato in Figura 40. Una funzione in `C` che esegue questa operazione è la seguente:

```
void addT(ListaDiElementi* l, int x){
    //creo un nuovo collegamento iniziale
    ListaDiElementi new = malloc(sizeof(ElementoDiLista));

    //il primo elemento contiene l'intero in input
    new->info = x;

    //collego il nuovo elemento a quello che era prima in testa
    new->next = *l;

    //modifico l
    *l = *new;
}

int main(){
    /* lista inizializzata in qualche modo */
    // lista = ...
    addT(&lista, 8);
}
```

Lo stato interno di questo programma è mostrato in Figura 41.

Esercizio 8.2 (Puntatore si oppure no)**.** La lista `l` viene passata alla funzione tramite puntatore per un motivo preciso, che a questo punto della dispensa dovrebbe essere chiaro. Si discuta di una definizione di `addT` con due parametri `ListaDiElementi l` e `int x` alla luce delle considerazioni fatte sui puntatori e la semantica blocchi.

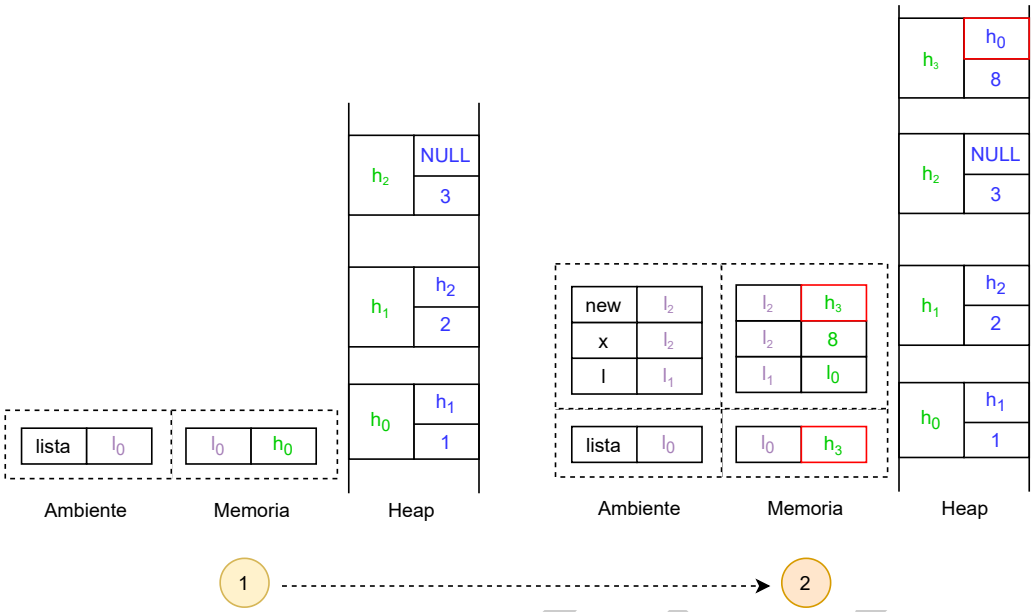


Figura 41. Stato interno di un programma che aggiunge un elemento in testa ad una lista. In rosso i cambiamenti salienti nel programma rispetto a XXXXX.

> **Nota bene** (Caso speciale: lista vuota). Si noti che `addT` funziona anche se la lista in input è vuota. Quindi si può utilizzare la funzione in questa maniera.
>
> ```
> int main(){
>     ListaDiElementi lista = NULL;
>
>     addT(&lista, 1);
>     addT(&lista, 2);
>     //e così via...
> }
> ```
>
> Si disegni la memoria dopo `addT(&lista, 1);` per convincersene.

Inserire elemento in coda alla lista. Come abbiamo inserito in testa, possiamo inserire in coda di una lista giá costruita. Per farlo bisogna distinguere due casi:

- se la lista in input è vuota, allora basterà usare il nuovo elemento in un *collegamento iniziale*;
- se la lista non è vuota, dovremo scorrere la lista fino a che non si trova un elemento il cui campo `next` è `NULL` - ovvero percorriamo la lista dal primo all'ultimo elemento - e creeremo un nuovo collegamento tra quest'ultimo e l'elemento, nuovo, che vogliamo aggiungere.

Entrambe le procedure sono qualitativamente mostrate in Figura 42. In codice definiamo una funzione `addC`:

```
void addC(ListaDiElementi* l, int x){
    //creo un nuovo elementi di lista contenente x
    ListaDiElementi new = malloc(sizeof(ElementoDiLista));
    (*new).info = x;
```

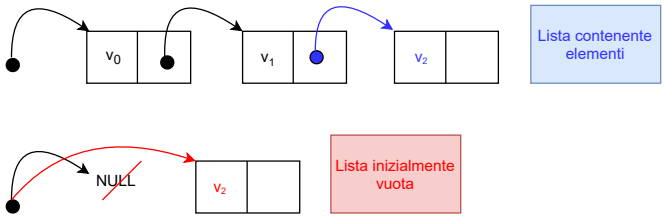


Figura 42. Rappresentazione schematica dell'aggiunta di un elemento in coda ad una lista, distinguendo due casi: lista non vuota, sopra, e lista vuota, sotto.

```
    (*new).next = NULL; // questo diventera il nuovo ultimo elemento

    //se la lista in input è vuota
    if(*l ==NULL)
        //puntatore tra il collegamento iniziale ed il nuovo elemento
        *l = new;
    else {
        //copio il puntatore alla lista in input
        ListaDiElementi corr = *l;

        //scorro tutta la lista fino a che non trovo l'ultimo elemento
        while(corr->next!=NULL)
            corr = corr->next;

        //sistemo i collegamenti per aggiungere elemento in coda
        corr->next = new;
    }
}
```

Nota bene. Dalla funzione `addC` possiamo riconoscere un pattern di programmazione utile tutte le volte che dobbiamo scorrere una lista linkata dal primo all'ultimo elemento.
Questo consiste nel definire un puntatore ausiliario ad una lista, in cui memorizzare l'indirizzo del primo elemento della lista in input (`corr`) e cambiare il valore di quest'ultimo assegnandogli l'indirizzo contenuto nel campo `next` dell'elemento corrente (ovvero farlo puntare al prossimo elemento nella lista). Siccome scorriamo una lista la cui dimensione è ignota, iteriamo con un `while` fino a che non viene trovato un puntatore a `NULL`, che segnala l'ultimo elemento della lista.

```
ListaDiElementi curr = *l;

//fino a che non trovo l'ultimo elemento
while(corr->next!=NULL)
    corr = corr->next;
```

Questo pattern è utile ad esempio per contare quanti elementi contiene la lista o quanto vale la somma degli elementi nella lista.

```
ListaDiElementi curr = *l;
```

```
int i = 0; //contatore per il numero di elementi
int somma = 0; //accumulatore per la somma degli elementi

while(corr->next!=NULL)
{
    i++;
    somma += corr->info;
    corr = corr->next;
}
printf("La lista contiene %d elementi la cui somma è %d", i, somma);
```

Si noti che se una procedura deve poter modificare il primo elemento della lista in input, allora il passaggio deve esser fatto per puntatore. Inoltre, se la procedura deve scorrere la lista, allora possiamo utilizzare un puntatore temporaneo ed un ciclo `while`.

Calcolare la lunghezza di una lista Vogliamo definire una funzione che calcoli la lunghezza di lunghezza di una lista. Questo può esser fatto in maniera iterativa nel seguente modo:

```
int length(ListaDiElementi l)
{
    int n=0; //num. di elementi
    while(l!=NULL)
    {
        n++;
        l = l->next;
    }
    return n;
}
```

Durante l'esecuzione di questo programma quello che facciamo è scorrere la lista dal primo all'ultimo elemento (caratterizzato dal fatto che `l->next==NULL`) tenendo il conto di quanti elementi stiamo attraversando. Tale numero di elementi è per definizione la lunghezza della lista.

La lunghezza di una lista può però esser calcolata anche in maniera ricorsiva:

```
int length_ricorsiva(ListaDiElementi l)
{
    //caso base: lista vuota
    if(l==NULL)
        return 0;
    //chiamata ricorsiva
    else
    {
        return 1 + length_ricorsiva(l->next);
    }
}
```

La logica di questo programma è che ogni volta si aggiunge al conto totale degli elementi della lista la testa della lista corrente e si effettua la chiamata ricorsiva sulla lista corrente senza l'elemento di testa.

Sommare gli elementi di una lista Data una lista linkata, si vuole definire una funzione che calcoli la somma degli elementi contenuti nella lista. Una possibile soluzione ricorsiva è la seguente:

```
int somma_lista(ListaDiElementi l)
{
    //caso base: ultimo elemento della lista
    //ovvero lista di un solo elemento
    if(l->next==NULL)
        return l->info;
    //chiamata ricorsiva
    else
    {
        return l->info + somma_lista(l->next);
    }
}
```

La logica di questo programma è la stessa dell'esempio del calcolo ricorsivo della lunghezza della lista: si scorre la lista dal primo all'ultimo elemento, accumulando il valore contenuto in `info` della lista corrente e effettuando la chiamata ricorsiva sulla lista corrente privata dell'elemento di testa.

Verificare se una lista è ordinata in modo non decrescente Vogliamo scrivere un programma che controlli se una data lista linkata è ordinata in modo non decrescente. Una possibile soluzione è la seguente:

```
int ord(ListaDiElementi l)
{
    int ordinato = 1; //flag, 1 se la lista è ordinata, 0 altrimenti
    //fino a che ho un elemento successivo e la lista è ordinata
    while(l->next != NULL && ordinato)
    {
        //se l'elemento puntato e il successivo contraddicono l'ordine non decrescente
        if(l->info<l->next->info)
            //modifico il flag
            ordinato = 0;
        l = l->next;
    }
}
```

Da notare che `l->next->info` si riferisce al campo `info` dell'elemento di lista successivo all'elemento corrente. Questo è il motivo per cui la prima condizione del comando `while` è `l->next!=NULL` e non, come abbiamo fatto negli esempi precedenti, `l!=NULL`.

Esercizio 8.3. Scrivere un programma che, dati in input una lista linkata `l`, un intero `x` ed un intero `v`, inserisce `v` nella lista prima della prima occorrenza di `x` in `l`. Se `x` non è presente in `l`, la lista non viene modificata. La lista viene cioè modificata solo se essa contiene `x`.

PART

II

Esercizi

9 Sintassi di base

Esercizi sulla sintassi di base del linguaggio `C`. Per risolverli si può partire da questo semplice template di programma.

```
// per avere printf e scanf
#include<stdio.h>

void main()
{
    /* programma qui */
}
```

Per compilare il programma (nel file `main.c`, assumendo il compilatore `gcc`), si utilizzino i seguenti comandi da shell

```
> gcc main.c -o main
> ./main
```

Esercizio 9.1. Si scriva un programma che legga due valori interi, calcoli la loro somma e media aritmetica, e le stampi a schermo.

Esempio di output.

```
> Inserisci il primo numero: 16
> Inserisci il secondo numero: 4
> La somma totale è: 20
> La media aritmetica è: 10
```

```
/* commenti etc etc. */
{
    int x = 10;
    int y = 1235;
}
```

Esercizio 9.2. Dato il seguente frammento di codice, disegnarne la memoria e stabilire quale valore assume `max`:

```
int x = 33;
int y = 10;
int max; //var. dove verrà memorizzato il massimo

/* condizionale */
if(x>y)
```

```
    max = x; //comando1
else
    max = y; //comando2
/* quanto vale max? */
```

Esercizio 9.3. Si scriva un programma che legga un valore intero e visualizzi il valore intero precedente e il successivo.

Esempio di output.

```
> Inserisci il numero: 5
> Il successivo di 5 è: 6
> Il precedente di 5 è: 4
```

Esercizio 9.4. Si scriva un programma capace di compiere le 4 operazioni (somma, sottrazione, moltiplicazione e divisione) tra due numeri reali inseriti da tastiera. Dopo che sono stati inseriti i due numeri, detti a e b, il programma dovrà visualizzare i quattro valori $a + b$, $a - b$, ab, a/b. Si ipotizzi che $b \neq 0$.

Esempio di output.

```
> Inserisci il primo numero (a): 16
> Inserisci il secondo numero (b): 4
> La somma a + b è: 20
> La differenza a - b è: 12
> Il prodotto a * b è: 64
> La divisone a / b è: 4
```

Esercizio 9.5. Determinare che cosa fa il seguente frammento di programma, disegnandone la memoria in modo opportuno.

```
int a, b, c;

scanf("%d", &a);
scanf("%d", &b);

if( a>b )
{
    c = a;
    a = b;
    b = c;
}

printf("%d\n", b) ;
```

Successivamente, scrivere un programma equivalente senza modificare a e b.

Esercizio 9.6. Si realizzi un programma che acquisisca da tastiera un numero x e stampi un messaggio che indichi se tale numero sia positivo oppure negativo, e ne stampi il valore assoluto $|x|$. Si richiede di risolvere l'esercizio senza alcuna variabile di appoggio.

Esempio di output.

```
> Inserisci il numero (x): -16
> Il numero 16 è negativo
> Il valore assoluto di -16 è |-16| = 16
```

Esercizio 9.7. Si scriva un programma che legga due numeri da tastiera, detti a e b, e determini le seguenti informazioni, stampandole a video:

- se b è un numero positivo o negativo;
- se a è un numero pari o dispari;
- il valore di $a + b$;
- quale scelta dei segni dell'espressione $\pm a + \pm b$ porta al risultato massimo, e quale è questo valore.

Suggerimento: il valore massimo della somma di a e b si può ottenere sommando il valore assoluto di a ed il valore assoluto di b.

Esercizio 9.8. Determinare il valore visualizzato dalle seguenti porzioni di codice nel caso in cui `num = 4` e per i seguenti valori della variabile conta: `conta = 5`, `conta = 0` e. `conta = 1`.

```c
// Programma 1
int conta, num;

scanf("%d", &conta); scanf("%d", &num);

while (conta != 0) {
    num = num * 10; conta = conta - 1;
}

printf("%d\n", num);
```

```c
// Programma 2

int conta, num;

scanf("%d", &conta); scanf("%d", &num);

while (conta > 0) {
    num = num * 10;
    conta = conta - 1;
}

printf("%d\n", num);
```

Esercizio 9.9 (Iterazione indefinita)**.** Si scriva un programma per calcolare la media aritmetica di una serie di numeri $x_1, \ldots, x_n$ inseriti da tastiera,

ovvero

$$x_* = \frac{1}{n} \sum_{i=1}^{k} x_i \, .$$

L'introduzione del valore 0 indica il termine del caricamento dei dati.

Esempio di output.

```
> Inserisci un numero (x_i): 10
> Inserisci un numero (x_i): 10
> Inserisci un numero (x_i): 5
> Inserisci un numero (x_i): 1
> Inserisci un numero (x_i): 15
> Inserisci un numero (x_i): 0
> La media dei numeri inseriti è = 8.2
```

```c
#include <stdio.h>

void main()
{
        // numero inserito da tastiera
        int x;
        float somma = 0;

        for (int i = 0; i < 3; i++)
        {
                printf("Inserisci il numero (x_%d): ", i);
                scanf("%d", &x);
                somma = somma + x;
        }

        printf("La media è: %f\n", somma / 3);
}
```

Esercizio 9.10 (Iterazione indefinita con min-max)**.** Si scriva un programma per calcolare il valore massimo e minimo di un insieme di n numeri inseriti da tastiera. Il programma deve prima leggere il valore di n dall'utente, ed in seguito leggere una sequenza di $n > 0$ numeri. A questo punto il programma deve stampare il massimo ed il minimo tra i numeri inseriti.

Esempio di output.

```
> Inserisci il numero (n): 4
> Inserisci il numero (1): 1
> Inserisci un numero (2): 11
> Inserisci un numero (3): 1
> Inserisci un numero (4): -4
> Il numero massimo inserito è = 11
> Il numero minimo inserito è = -4
```

Esercizio 9.11. Dati due interi positivi n e k da tastiera, calcolare la sommatoria:

$$S(n,k) = \sum_{i=1}^{n} k^i = k + k^2 + \cdots + k^n$$

e stampare a schermo il risultato. Ad esempio, con $n = 2$ e $k = 3$ allora $S(2,3) = 3 + 3^2 = 3 + 9 = 12$.

Esempio di output.

```
> Inserire n: 2
> Inserire k: 3
> Risultato sommatoria: 12
```

```c
#include<stdio.h>

int main() {
  int n, k; // Chiediamo `n` e `k` all'utente
  printf("Inserire n: ");
  scanf("%d", &n);
  printf("Inserire k: ");
  scanf("%d", &k);
  int somma = 0; // Variabile che conterrà il risultato
  int tmp = k; // Variabile ausiliaria

  for(int i = 1; i <= n; i++) { // Ciclo sugli elementi della sommatoria
    somma += tmp; // Aggiungo `k ^ i` alla somma
    tmp = tmp * k; // Calcolo `k ^ (i + 1)`
  }

  printf("Risultato sommatoria: %d\n", somma);
  return 0;
}
```

Esercizio 9.12. Letto un intero positivo n da tastiera, stampare tutta la successione di Fibonacci fino all'elemento n-esimo compreso. Chiamando $F(i) = F_i$ l'elemento i-esimo della successione di Fibonacci allora:

$$F_i = \begin{cases} 0 \text{ se } i = 0, \\ 1 \text{ se } i = 1, \\ F_{i-1} + F_{i-2} \text{ se } i > 1 \end{cases}$$

ad esempio, considerando $i = 3$, $F_3 = F_2 + F_1 = (F_1 + F_0) + F_1 = 2$.

Esempio di output.

```
> Inserire n: 3
> F(0) = 0
> F(1) = 1
```

```
> F(2) = 1
> F(3) = 2
```

```c
#include <stdio.h>

int main() {
  int n;
  printf("Inserire n: ");
  scanf("%d", &n);

  // Inizializzo e stampo i primi due valori della successione
  int F0 = 0, F1 = 1;
  printf("F(0) = %d\n", F0);
  printf("F(1) = %d\n", F1);

  // Se `i > 1` inizializzo gli elementi `i - 1`-esimo e `i - 2`-esimo
  int Fi, Fi_1 = F1, Fi_2 = F0;

  for(int i = 2; i <= n; i++) {
    Fi = Fi_1 + Fi_2; // Uso la formula per calcolare l'elemento `i`-esimo
    printf("F(%d) = %d\n", i, Fi);
    Fi_2 = Fi_1; // Aggiorno per calcolare l'elemento `i + 1`-esimo
    Fi_1 = Fi;
  }

  return 0;
}
```


Esercizio 9.13. Dati due interi positivi n e k da tastiera, calcolare il coefficente binomiale di n su k. Il coefficente binomiale $C(n,k)$ di n su k, anche indicato con $\binom{n}{k}$, è dato da:

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

ad esempio, con $n = 3$ e $k = 2$:

$$C(3,2) = \frac{3!}{2! \cdot 1!} = \frac{6}{2} = 3$$

nota che $C(n,k)$ è definito per $n \geq k$ ma può essere esteso ponendo $C(n,k) = 0$ se $n < k$.

Esempio di output.

```
> Inserire n: 3
> Inserire k: 2
> Coefficente binomiale di 3 su 2 è: 3
```

```c
#include <stdio.h>

int main() {
  int n, k; // Chiediamo `n` e `k` all'utente
  printf("Inserire n: ");
  scanf("%d", &n);
  printf("Inserire k: ");
  scanf("%d", &k);
  int coeff_bin = 0;

  if(n >= k) { // Se `n` è maggiore o uguale a `k` uso la formula altrimenti rimane 0
    int n_fact = 1;
    int k_fact = 1;
    int nk_fact = 1;

    for (int i = n; i > 0; i--) {
        if (i <= k)
            k_fact *= i; // Calcolo `k!`

        if (i <= (n - k))
            nk_fact *= i; // Calcolo `(n - k)!`

        n_fact *= i; // Calcolo `n!`
    }

    coeff_bin = n_fact / (k_fact * nk_fact); // Uso la formula
  }

  printf("Coefficente binomiale di %d su %d: %d\n", n, k, coeff_bin);
  return 0;
}
```


10 Funzioni iterative e ricorsive

Esercizio 10.1 (Minimo)**.** Si scriva una funzione che riceva in ingresso tre numeri interi x, y e z e ne restituisca il minimo. Si crei una funzione di test che prenda in input due dei tre numeri dall'utente, x ed y, e testi la tripletta (x, y, w) con $w \in [1, \max\{x, y\}]$.

Nota: per testare la tripletta si intende testare ogni possibile tripla di valori ottenuta fissando x ed y, e variando w.

Esempio di output.

```
> Inserisci il numero (x): 1
> Inserisci un numero (y): 3
> Inserisci un numero (z): 4
> tripletta (1, 2, 1) - minimo -> 1
> tripletta (1, 2, 2) - minimo -> 1
> tripletta (1, 2, 3) - minimo -> 1
```

Esercizio 10.2 (Potenza)**.** Si scriva una funzione che riceva in ingresso due numeri interi a e b ($b > 0$) e restituisca il risultato della potenza a^b, in modo iterativo e ricorsivo. Si crei una funzione di test che testa tutte le possibili coppie di valore nell'intervallo $[1, 5]$.

Esempio di output (restringendo l'intervallo di test a $[1, 2]$).

```
> Inserisci il numero (a): 2
> Inserisci un numero (b): 3
> a^b : 8
> coppia (a = 1, b = 2) - a^b = 1^2 = 1
> coppia (a = 2, b = 2) - a^b = 2^2 = 4
> coppia (a = 2, b = 1) - a^b = 2^1 = 2
> coppia (a = 2, b = 2) - a^b = 2^2 = 4
```

```c
// Calcola ricorsivamente la potenza

#include<stdio.h>

long int potenza(int a, int b) {
  long int power = 1; // Tiene traccia della potenza calcolata

  if(b == 0) { // Base ricorsione
    // `power` è 1 qui
    return power;
  }

  power = a * potenza(a, b - 1); // Ricorsione
}

int main() {
  int base, esp; // Base e esponente della potenza
  printf("Inserire base: ");
  scanf("%d", &base);
  printf("Inserire esp: ");
  scanf("%d", &esp);
  long int result = potenza(base, esp);
  printf("%d^%d = %ld\n", base, esp, result);

  return 0;
}
```

Esercizio 10.3 (Norma)**.** Si scriva una funzione che, ricevuti in ingresso le coordinate $p_1 = (x_1, y_1)$ ed $p_2 = (x_2, y_2)$ di due punti del piano cartesiano, restituisca la loro distanza euclidea

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\,,$$

Nota: la funzione radice quadrata (`sqrt`) in C viene fornita dalla librera `math.h`, che va quindi importata come segue:

```c
#include<stdio.h> // input output
#include<math.h> // sqrt

double distanza(double x1, double y1, double x2, double y2) {
  // codice vostro
  ....
}
```

Esercizio 10.4 (Uguaglianza approssimata)**.** Si scriva una funzione che riceva in ingresso due numeri `double` x ed y e restituisca 1 se e solo se i due numeri sono uguali a meno di un fattore $\epsilon = 10^{-9}$, ovvero

$$|x - y| < \epsilon$$

e 0 in caso contrario. Il valore di ϵ deve essere un parametro della funzione.

```
> Inserisci epsilon (e): 0.000001
> Inserisci il numero (x): 0.001
> Inserisci un numero (y): 1.001
> i due numeri sono diversi con epsilon 0.000001
```

Esercizio 10.5. Si scriva una funzione che riceva in ingresso un numero `float` corrispondente al prezzo iniziale di un articolo e un numero `int` rappresentante il valore percentuale di sconto e restituisca il prezzo scontato.

Esercizio 10.6 (Fattoriale)**.** Si consideri il *fattoriale*

$$n! = n(n-1)!$$

definito per $n > 0$ intero.

- Si implementi la definizione iterativa del fattoriale;
- Si implementi la definizione ricorsiva del fattoriale;
- si crei un programma che testa entrambe le versioni per i valori di input che vanno da 1 a 10. Il programma deve stabilire se entrambe le funzioni danno lo stesso risultato sempre, e stampare un messaggio adeguato.

Esempio di output (restringendo a valori minori di 4).

```
> Fattoriale ricorsivo/iterativo di 1: 1/1
> Fattoriale ricorsivo/iterativo di 2: 2/2
> Fattoriale ricorsivo/iterativo di 3: 6/6
> Fattoriale ricorsivo/iterativo di 4: 24/24
> Le due funzioni danno lo stesso risultato sempre: vero
```

Esercizio 10.7 (Somma naturali)**.** Si consideri la seguente somma dei primi n numeri naturali

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

definita per $n > 0$ intero.

- Si implementi la definizione iterativa della somma;
- Si implementi la definizione ricorsiva della somma;
- Si implementi la soluzione esatta per la somma;
- si crei un programma che testa le tre versioni per i valori di input che vanno da 1 a 100. Il programma deve stabilire se tutte le funzioni danno lo stesso risultato sempre, e stampare un messaggio adeguato.

Esempio di output (restringendo a valori minori di 4).

```
> Somma ricorsivo/iterativo/esatta fino a 1: 1/1/1
> Somma ricorsivo/iterativo/esatta fino a 2: 3/3/3
> Somma ricorsivo/iterativo/esatta fino a 3: 6/6/6
> Somma ricorsivo/iterativo/esatta fino a 4: 10/10/10
> Le tre funzioni danno lo stesso risultato sempre: vero
```

```c
// Somma dei primi `n` naturali ricorsiva

#include <stdio.h>

int sum(int n) {
  int partial = n; // Variabile contenente la somma parziale da 1 ad `n`: la inizializzo a `n`

  if(n == 1) // Base della ricorsione: ho esaurito gli interi
        return 1;

  partial += sum(n - 1); // Calcolo la somma parziale aggiungendo quella di `n - 1` interi

  return partial;
}

int main() {
  int n;
  printf("Inserire un numero intero positivo: ");
  scanf("%d", &n);
  printf("La somma dei primi %d naturali è: %d\n", n, sum(n));
  printf("Controllo: %d\n", n * (n + 1) / 2);

        return 0;
}
```

Esercizio 10.8 (Intervallo di numeri)**.** Si considerino due interi $a > 0$, $b > 0$, $a < b$. Si scriva una funzione che visualizza, tramite `printf`, i numeri nell'intervallo generalizzando la seguente regola

$$a = 1, b = 6 \rightarrow 1, 3, 5$$
$$a = 2, b = 11 \rightarrow 2, 4, 6, 8, 10$$

Si noti che la funzione non restituisce alcune valore. Si testi la funzione su valori di a e b dati in input dall'utente mediante `scanf`.

Esercizio 10.9 (Somma di cifre)**.** Dato un numero $n \in \mathbb{N}$ espresso come sequenza di $w > 0$ cifre

$$n \equiv c_1 c_2 c_3 c_4 \cdots c_w$$

si definisca la quantità

$$d = \sum_{i=1}^{w} c_i$$

- Si implementi la definizione iterativa per il calcolo di d;
- Si implementi la definizione ricorsiva per il calcolo di d;
- si crei un programma che testa le due versioni per i valori di input che vanno da 1 a 1000 (incluso). Il programma deve stabilire quale numero ha il valore di d associato più alto, e stampare un messaggio adeguato.

Esempio di output (alcuni valori a campione).

```
> ....
> Somma cifre ricorsiva/iterativa 234: 9/9
> Somma cifre ricorsiva/iterativa 235: 10/10
> Somma cifre ricorsiva/iterativa 236: 11/11
> ....
> Il valore massimo per la ricorsiva é 27 per il numero 999.
> Il valore massimo per la iterativa é 27 per il numero 999.
```

```c
// Somma cifre numero intero positivo ricorsiva

#include <stdio.h>

int sum(int n) {
  int tmp = n % 10; // Prendo la cifra delle unità

  if(n == 0) // Base della ricorsione: ho esaurito tutte le cifre
  return 0;

  return tmp + sum(n / 10); // Ricorsione eliminando la cifra delle unità
}
```

```c
int main() {
  int n;
  printf("Inserire un numero intero positivo: ");
  scanf("%d", &n);
  printf("La somma delle cifre di %d è: %d\n", n, sum(n));

  return 0;
}
```

Esercizio 10.10 (Numero primo)**.** Dato un numero $n \in \mathbb{N}$, $n > 0$ vogliamo stabilire se sia primo oppure no.

- Si implementi la definizione iterativa per il calcolo del predicato n è un numero primo;
- Si implementi la definizione ricorsiva per il calcolo del predicato n è un numero primo;
- si crei un programma che testa le due versioni per i valori di input che vanno da 1 a 100 (incluso). Il programma deve stabilire il numero di numeri primi trovati nell'intervallo, e la distanza media tra ciascuno dei numeri primi trovati. Ad esempio, la distanza media tra 1, 3 e 5 vale $[(3-1)+(5-3)]/2$.

Si noti che le due funzioni potrebbero non avere gli stessi argomenti.

Esempio di output (su un sotto-insieme di valori).

```
> Test ricorsivo/iterativo primo 1: vero/vero
> Test ricorsivo/iterativo primo 3: falso/falso
> Test ricorsivo/iterativo primo 5: vero/vero
> Ci sono 2 numeri primi nell'intervallo [1, 5]
> La distanza media tra essi é 2
```

```c
 // Stabilire se un numero è primo ricorsivamente

#include <stdio.h>

int is_prime(int n, int m) {
        if (m == 1) // Base della ricorsione: il divisore è 1 allora il numero è primo
                return 1;
        else if (n % m == 0 && n != m) // Oppure il numero è divisibile per un numero diverso da sè stesso
                return 0; // Allora il numero non è primo

        is_prime(n, m - 1); // Provo con un altro divisore
}

int main() {
        int n, m = 9;
  printf("Inserire un numero intero positivo: ");
```

```
  scanf("%d", &n);

  if (is_prime(n, m) == 1)
          printf("%d è primo\n", n);
  else
    printf("%d non è primo\n", n);

        return 0;
}
```

Esercizio 10.11 (GCD)**.** Si scriva l'algoritmo di Euclide per il calcolo del GCD utilizzando una definizione ricorsiva `gcd(a,b)`, con $a, b > 0$ ed interi. Si testi la propria implementazione con numeri dati in input dall'utente, confrontando il risultato ottenuto con la versione iterativa discussa in questa dispensa e vista a lezione.

Esercizio 10.12 (LCM)**.** Si consideri la definizione di *Least Common Multiple* (LCM) - minimo comune multiplo. Si scriva una funzione ricorsiva `lcm(a,b)` per il calcolo dell' LCM a partire da due valore $a, b > 0$ ed interi. La si verifichi tenendo conto che `lcm(a,b)=ab/gcd(a,b)` con numeri dati in input dall'utente, dove `gcd` é la funzione definita nel precedente esercizio.

Esercizio 10.13 (Fibonacci)**.** Si consideri la *successione di Fibonacci*

$$F_n = \begin{cases} 1 & \text{se } n = 1; \\ 1 & \text{se } n = 2; \\ F_{n-1} + F_{n-2} & \text{altrimenti} \end{cases}$$

definita per $n > 0$ intero.

- Si implementi la definizione iterativa per il calcolo di F_n;
- Si implementi la definizione ricorsiva per il calcolo di F_n;
- si crei un programma che testa le due versioni per i valori di input che vanno da 1 a 10 (incluso).
- si testino entrambe le funzioni per $n = 100$, e si provi a spiegare il comportamento osservato.

Esempio di output (su un sotto-insieme di valori).

```
> Fibonacci ricorsivo/iterativo  1: 1/1
> Fibonacci ricorsivo/iterativo  2: 1/1
> Fibonacci ricorsivo/iterativo  3: 2/2
```

```
// Successione di Fibonacci ricorsiva

#include <stdio.h>
```

```c
void fibonacci(int a, int b, int n) {
  int sum; // Memorizza somme parziali

  if(n > 0) {
    sum = a + b; // Aggiorna somma parziale
    printf("%d, ", sum);
    a = b; // Aggiorna per la chiamata ricorsiva
    b = sum;
    fibonacci(a, b, n - 1); // Ricorsione
  }
}

int main() {
  int a = 0, b = 1; // Primi due termini della serie
  int n;
  printf("Inserire numero totale di termini: ");
  scanf("%d", &n);
  printf("La serie di Fibonacci è: \n");

  // Stampo i primi due termini
  printf("%d, %d, ", a, b);

  // Chiamo la funzione per gli `n - 2` rimanenti termini
  fibonacci(a, b, n - 2);
  printf("...\n");

  return 0;
}
```

Esercizio 10.14 (Hailstone)**.** Si consideri la *sequenza di Hailstone*: si parta da un intero $n > 0$

- si generi il prossimo numero m della sequenza come segue:
  - se n è pari, allora $m = n/2$;
  - altrimenti, $m = 3n + 1$;
- si ripeta fino a che non si genera $m = 1$;
- si conti la lunghezza della sequenza generata.

Si utilizzino due funzioni - sequenza e lunghezza - per la definizione della suddetta sequenza a partire da un intero $n > 0$ ricevuto in input dall'utente.

Esercizio 10.15 (Conversione di base)**.** Vogliamo convertire un numero da una base numerica all'altra. L'algoritmo per convertire un intero $n > 0$ da base decimale a base binaria è il seguente:

1. calcolare la divisione intera $n' = n/2$ con resto r';

2. r' è la cifra più a destra nella rappresentazione binaria di n, chiamiamola c_0;
3. si divide ancora $n'' = n'/2$ con resto r'' ed otteniamo la seconda cifra $r'' = c_1$;
4. si ripete il procedimento fino a quando il risultato della divisione è uguale a 0, ottenendo la rappresentazione binaria di n concatenando le cifre c_i come $c_k c_{k-1} \cdots c_0$.

Mentre l'algoritmo inverso si basa sul fatto che che

$$n = c_0 2^0 + c_1 2^1 + \ldots c_k 2^k$$

- scrivere una funzione che, preso in input un numero intero in base decimale, lo converta in base binaria;
- scrivere la funzione di conversione inversa;
- testare le funzioni cosi definite sui numeri da 1 a 100.

Esempio di output (su un sotto-insieme di valori).

```
> b10 -> b2 -> b10 = 0 -> 0 -> 0
> b10 -> b2 -> b10 = 1 -> 01 -> 1
> b10 -> b2 -> b10 = 2 -> 10 -> 2
> b10 -> b2 -> b10 = 3 -> 011 -> 3
> b10 -> b2 -> b10 = 4 -> 100 -> 4
```

Nota: Non sapendo ancora come memorizzare sequenze di bits 0/1 di lunghezza arbitraria, si consideri che per convertire un numero < 100 in binario servono al più $\log_2(100)$ bits.

11 Serie e successioni

> **Nota bene.** Tutte le successioni o serie negli esercizi qui proposto devono essere implementate in modo sia iterativo che ricorsivo. Per ciascun calcolo implementato si verifichi la congruenza trai i valori restituit,i almeno per un insieme di input ragionevoli (e.g., i primi 50 termini della successione/serie).

Per tutti gli esercizi:

Esempio di output.

```
> s_1 ricorsivo/iterativo = 2.24431/2.24431
> s_2 ricorsivo/iterativo = 2.5445/2.5445
> s_3 ricorsivo/iterativo = 5.531/5.531
> ....
```

Nota: si ponga particolare attenzione al tipo di valori ritornati, in particolare alla differenza tra ritornare un `float` e non un `int`, ove opportuno.

Esercizio 11.1. Si implementi una funzione che calcola la seguente successione fino al termine n-esimo:

$$a_n = \begin{cases} \frac{1}{2} & \text{se } n = 1; \\ \frac{a_{n-1} + 1}{2} & \text{altrimenti} \end{cases}$$

Verificare che il limite della successione è 1.

Esercizio 11.2. Si implementi una funzione che calcola la seguente successione fino al termine n-esimo:

$$a_{n+1,p} = \begin{cases} p, & n = 1 \\ \frac{1}{2}(a_{n,p} + \frac{p}{a_{n,p}}) & n > 1 \end{cases} \tag{11.1}$$

dove p è un parametro costante della successione.

Verificare che il limite della seguente successione è $\sqrt{p}$.

Esercizio 11.3 (Serie geometrica)**.** Si consideri la seguente serie geometrica:

$$s_n = \sum_{k=0}^{n} x^k \,, \tag{11.2}$$

dove x è un valore scelto dall'utente. Verificare che la serie equivale a:

$$s_n = \frac{1 - x^{n+1}}{1 - x} \tag{11.3}$$

Esercizio 11.4. Si consideri la seguente serie:

$$s_n = \sum_{k=0}^{n} \frac{1}{(2k+1)^2} \tag{11.4}$$

Verificare che per $n \to \infty$ la serie tende a $\frac{\pi^2}{8}$.

Nota: Testare la propria implementazione per valori abbastanza grandi di n; anche se questa non è una prova generale, in questo caso il controllo basta per verificare la convergenza.

Esercizio 11.5 (Numeri altamente composti)**.** Scrivere un programma per calcolare i primi k numeri altamente composti, dove il numero k è definito dall'utente (la scelta della modalità di scrittura è libera). Un numero altamente composto è un intero positivo che ha più divisori di qualsiasi intero positivo

minore - la definizione esatta di tali numeri si trova banalmente su Wikipedia (https://it.wikipedia.org/wiki/Numero_altamente_composto).

Soluzione: 1, 2, 4, 6, 12, 24, 36, 48, 60, 120, 180, 240, 360, 720, 840, ...

12 Puntatori e Array

In alcuni di questi esercizi dovrete realizzare, oltre alle funzioni richieste dall'esercizio, un metodo `main` che testerà il calcolo implementato. Il metodo potrà allocare staticamente un array di test per l'esercizio, oppure caricare l'input dopo una interazione con l'utente (tramite `scanf`).

Nota: per la risoluzione di un esercizio potreste scegliere di implementare un numero di funzioni ausiliarie a scelta vostra, se opportuno.

Esercizio 12.1 (Varianza)**.** Si scriva una funzione che prenda in input un array di interi e restituisca la varianza

$$\sigma = \frac{\sum(x_i - \mu)^2}{n} \tag{12.1}$$

degli n lementi in esso contenuti; nella formula x_i è l'elemento i-esimo dell'array mentre μ è la media degli elementi dell'array.

Richiesta: La segnatura della funziona sarà

```
float var(int a[], int n)
```

Nella prima soluzione le funzioni prendono come argomento un array di interi `a`, mentre nella seconda soluzione l'argomento `a` è un puntatore.

Prima soluzione.

```c
#include <stdio.h>

// funzione per calcolare la media di un array "a"
double media(int a[], int n)
{
        double somma = 0;
        for (int i = 0; i < n; i++)
                somma += a[i]; // somma = somma + a[i]

        return (somma / n);
}

// funzione per calcolare la potenza di un numero "n"
double pow2(double n)
{
        return n * n;
}

// funzione per calcolare la varianza degli elementi di un array "a"
```

```
double varianza(int a[], int n)
{
        double m = media(a, n);
        double num = 0;

        for (int i = 0; i < n; i++)
                num += pow2(a[i] - m);

        return (num / n);
}
```

Seconda soluzione.

```
#include <stdio.h>

double media(int *a, int l)
{
        double s = 0;
        for (int i = 0; i < l; i++)
                s += *(a + i);

        return s / l;
}

double pow2(double n)
{
        return n * n;
}

double varianza(int *a, int l)
{
        double av = media(a, l);

        double s = 0;
        for (int i = 0; i < l; i++)
                s += pow2(*(a + i) - av);

        return s / l;
}
```

Il main è equivalente per entrambe le soluzioni.

```
// funzione ausiliaria per stampare i valori di un array
void print_array(int a[], int N)
{
        for (int i = 0; i < N; i++)
                printf("a[%d] = %d\n", i, a[i]);
        printf("\n");
}

void main()
{
        int n = 5;
        int a[5] = {1, 2, 3, 4, 5};
```

```
        print_array(a, n);

        double var = varianza(a, n);
        printf("La varianza dell'array è %f\n", var);
}
```

Esercizio 12.2. Si scriva una funzione che data la somma e la differenza delle basi di un trapezio ne calcoli l'area, sapendo che l'altezza è $\frac{2}{3}$ della base minore. La funzione dovrà avere la segnatura:

```
void area(int somma, int differenza, float* area
```

Esercizio 12.3. Secondo il linguaggio C, `a[0]` equivale in realtà a scrivere `*(a+0)`. Si scriva un programma che stampi tutti gli elementi di un array usando la seconda notazione.

Esercizio 12.4 (Stampa indirizzi)**.** Si scriva una funzione che prenda in input una variabile `int` e ne stampi l'indirizzo. Successivamente, si dichiari nel `main` una variabile, stampandone l'indirizzo e poi passandola alla funzione descritta sopra. I due indirizzi stampati sono gli stessi? Si provi a spiegare quanto osservato, aiutandosi disegnando la memoria del programma realizzato.

Esercizio 12.5 (Inversione di array)**.** Si scriva una funziona che legga un array di interi di dimensione fissata, lo inverta e lo stampi.

Nota: l'inversione deve essere esplicita, non basta stampare l'array in senso inverso (i.e., dagli ultimi ai primi elementi).

Esercizio 12.6 (Tutti uguali)**.** Si scriva una funzione che legga un array di lunghezza arbitraria e determini se l'array contiene solo valori diversi, o no.

Esercizio 12.7 (Ordinamento)**.** Si scriva una funzione che prenda un input un array e permuti i suoi elementi in modo che l'array risulti ordinato in maniera crescente.

Esercizio 12.8. Si scriva una funzione che prenda in input tre array $A1$, $A2$ ed $A3$, della stessa dimensione. La funzione, per ogni i calcola la somma tra $A1[i]$ ed $A2[i]$, e ne salva il risultato in $A3[i]$.

Scrivere un `main` che inizializzi i due vettori $A1$ e $A2$, invochi la funzione di somma ed infine stampi il contenuto di $A3$. Nella funzione si richiede di utilizzare il passaggio di parametri per indirizzo.

Esercizio 12.9 (Numero triangolare)**.** Si definisce *triangolare* un numero costituito dalla somma dei primi n numeri interi positivi per un certo n. Ad esempio per $Q = 10$ si ha $Q = 1 + 2 + 3 + 4$ da cui $n = 4$.

Si scriva una funzione che chieda all'utente n, ed inserisca in un array di dimensione n i primi n numeri triangolari.

Esercizio 12.10 (Divisore di tutti)**.** Si scriva una funzione che riceva in input un array di interi e la sua lunghezza, e restituisca 1 se uno degli elementi contenuti è divisore proprio di tutti gli altri, 0 altrimenti.

Esercizio 12.11 (Gluing)**.** Si scriva un programma che legga due stringhe da tastiera, le concateni in un'unica stringa e stampi la stringa così generata.

Esercizio 12.12 (Palindromo)**.** Si scriva una funzione che prenda in input una stringa e restituisca 1 se la stringa è palindroma, 0 altrimenti. Esempi di stringhe palindrome sono `anna`, `oro`, etc., ovvero stringhe che che sono leggibili in ugual modo da sinistra a destra, e viceversa.

Nota: si implementi anche una funzione che esegua lo stesso test a livello di array di interi. In tal caso si intende palindromo il vettore `[1,2,3,3,2,1]` e non il vettore `[1,2,3]`.

PART

III

Principi di programmazione ad oggetti

13 Introduzione a `Python`

Per una introduzione a `Python` – nello specifico `Python3` – riguardante costrutti di base, sintassi dei blocchi e cicli, dichiarazione di variabili e tipi, definizioni di funzioni, etc. si faccia riferimento alla dispensa di Laboratorio di Programmazione.

14 Programmazione ad oggetti

14.1 Classi ed oggetti

> **Nota bene.** L'argomento "classi" viene trattato anche nella dispensa di Laboratorio di Programmazione. Tuttavia, a differenza di altri concetti, viene qui ripreso al fine di poter approfondire, in modo sistematico, il concetto di ereditarietà in un paradigma ad oggetti nella Sezione 14.2.

Il concetto di **classe** è alla base della **programmazione ad oggetti**, paradigma di programmazione a cui `Python` appartiene. Altri linguaggi di programmazioni ad oggetti, quali `Java` o `C++`, mantengono ovviamente il concetto di classe, seppur con lievi differenze legate prevalentemente alla parte imperativa del linguaggi (ad esempio, in `C++` ci sono i puntatori espliciti, che in `Java` sono solamente impliciti).

Una classe in `Python` rappresenta, in modo abbastanza intuitivo, l'equivalente della `struct` in `C`, e si usa per costruire **oggetti**. In particolare, come la `struct` serve a definire un tipo, raccogliendo quindi la definizione delle strutture dati (campi, o valori) che rappresentano l'entità concettuale che vogliamo modellare - ciascun ogetto della classe rappresenta una realizzazione indipendente del concetto modellato dalla classe. A differenza della `struct`, invece, la classe non raccoglie solamente valori, ma anche funzioni - o metodo della classe - per operare sulle entità che modelliamo.

La sintassi per definire una classe, nel caso più semplice possibile, è la seguente:

```
class class_name:
    docstring

    .. variabili ..

    .. funzioni ..
```

dove:

- `class_name` è il nome della classe;
- `variabili` rappresentano variabili di istanza di ciascun oggetto che viene costruito;
- `funzioni` rappresentano funzioni associate (cioè implementate) per tutti gli oggetti della classe.

Ad esempio

```
class lamiaclasse:
    "Ho creato una classe"

    # funzione definita all'interno della classe
```

```
    def fun(self):
        print("Sto stampando da dentro la classe")
```

Nota bene (self)**.** Il parametro `self` nel metodo `fun` è una variabile che indica l'istanza corrente della classe, ovvero l'oggetto specifico su cui il metodo `fun` viene invocato. Questo `fun` in realtà non usa mai `self`, ma il passaggio di `self` a `fun` è obbligatoria. Il concetto espresso da `self` è più o meno presente in tutti i linguaggi ad oggetti, ad esempio in `Java` il `self` non è altro che il `this`.

Proviamo a mettere in pratica la definizione di classe per descrivere gli studenti dell'Università di Trieste.

```
class Studente:
    # attributo uguale per tutti
    ruolo = "Studente UNITS"

    # costruttore, per ogni studente,
    # che permette di creare un oggetto
    # della classe, con opportuni valori per
    # eventuali variabili di istanza dell'oggetto Studente
    def __init__(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def bonjour(self):
        print(self.ruolo, ":", self.nome, self.cognome)
```

Nel caso particolare un oggetto di classe `Studente` ha 3 attributi:

- `nome` e `cognnome`, che vengono settati su `self` all'interno di `__init__`;
- `ruolo`, un attributo uguale per tutte le istanze della classe `Studente`.

Questo esempio ci chiarisce cosa significhi creare un oggetto di una classe. Quando viene creato un oggetto di una certa classe, si istanzia un particolare elemento della classe di appartenenza. Questo avviene attraverso il metodo `__init__`, ovvero il metodo *costruttore* della classe. `__init__` è un particolare tipo di metodo che appartiene ai cosiddetti *metodi magici* (magic methods o dunder).

Nota bene (Magic methods)**.** I magic methods sono particolari metodi implementabili nelle classi Python, che permettono di far sì che le nostre classi possano interagire con alcuni operatori e costrutti del linguaggio. Tutti i cosiddetti magic methods hanno nome in forma `__method__` caratterizzata da underscores doppi (`__`).

Esempio 14.1 (String method)**.** Il metodo magico `__str__` si usa per creare una rappresentazione in stringa dell'oggetto di una classe. Tale metodo viene invocato ogni volta che si chiama o una `print()` o una `str()` su un oggetto della classe.

```
class A:
```

```
    x = 0
    y = ""

    def __init__(self, n, s):
        self.x = n
        self.y = s

myObject = MyClass(12345, "Hello")

print(myObject.__str__()) # <__main__.MyClass object at 0x7f66d12bd1f0>
print(myObject.__repr__()) # <__main__.MyClass object at 0x7f66d12bd1f0>
print(myObject) # <__main__.MyClass object at 0x7f66d12bd1f0>
```

Usando il metodo magico `__str__`

```
class A:
    x = 0
    y = ""

    def __init__(self, n, s):
        self.x = n
        self.y = s

    def __str__ (self):
        return 'A(x=' + str(self.x) + ' , y=' + self.y + ')'

myObject = A(12345, "Hello")

print(myObject.__str__()) # A(x=12345 , y=Hello)
print(myObject) # A(x=12345 , y=Hello)
print(str(myObject)) # A(x=12345 , y=Hello)
```

Quindi `__init__` permette di creare lo stato interno di una particolare istanza di un oggetto, un'operazione che sostanzialmente comporta l'assegnamento di un valore agli attributi che caratterizzano l'oggetto stesso. In questo caso quindi ogni oggetto di tipo `Studente` viene creato con gli attributi `nome, cognome, ruolo`. Si noti quindi che ogni oggetto avrà i suoi valori di `nome` e `cognome`, ma invece `ruolo` risulterà costante per tutti gli oggetti della classe.

Si noti che stavolta il metodo `bonjour`, ricevendo in input un particolare oggetto, utilizza `self.ruolo` e `self.nome` etc. per accedere al valore degli attributi dell'oggetto (indipendentemente dal fatto che siano stati creati nel costruttore oppure no).

Esercizio 14.1. Non affrontiamo un discorso sulla memoria in Python. Tuttavia, si ragioni ad alto livello su come si rappresenterebbe la memoria di un programma che crea oggetti della classe `Studente`.

Si noti che i vincoli descritti precedentemente riguardanti il parametro `self`, si applicano anche al costruttore della classe (ed agli altri metodi magici).

Un possibile utilizzo di questa classe è il seguente:

```
# creo un oggetto e chiamo la funzione bonjour
```

```
obj_Giulio = Studente("Giulio", "Caravagna")
obj_Giulio.bonjour()

# accedo al campo nome (come nelle struct in C)
print("Campo nome di Giulio", obj_Giulio.nome)
```

Si noti anche che i campi di un oggetto cosi definiti sono modificabili (come nelle `struct` in C).

```
obj_Giulio.nome = "Giuliano"  # cambio nome
print("Campo nome di Giulio", obj_Giulio.nome)

obj_Giulio.ruolo = "Bidello" # cambio ruolo
obj_Giulio.bonjour()
```

Si provi ad inserire il codice qui sopra per osservare il comportamento dell'interprete Python.

Esercizio 14.2 (Modellazione a classi)**.** Si elenchino almeno 5 contesti di applicazione per la programmazione, in cui potremmo beneficiare di introdurre delle classi, e si discuta di cosa modellerebbero i relativi oggetti. Per esempio: software di gestione gestione di un zoo, in cui ogni animale è rappresentato dalla sua classe, e gli oggetti sono gli specifici animali.

14.2 Ereditarietà tra classi

L'**ereditarietà** è uno dei concetti fondamentali nel paradigma di programmazione a oggetti, e consiste in una relazione che il programmatore stabilisce tra due classi.

Se la classe `B` eredita dalla classe `A`, si dice che `B` è una *sotto-classe* di `A` e che `A` è una *super-classe* di `B`. A seconda del linguaggio di programmazione, l'ereditarietà può essere singola (ogni classe può avere al più una super-classe) o multipla (ogni classe può avere più super-classi). In Python entrambe sono possibili, ma nel nostro caso ci limitermo a discutere l'ereditarietà singola. Si noti inoltre che la relazione è transitiva: in un programma potrei avere `A` super-classe di `B`, e `B` super-classe di `C`.

> **Nota bene** (Interpretazione modellistica dell'ereditarietà)**.** L'ereditarietà è una relazione di generalizzazione e specializzazione: la super-classe definisce un concetto generale e la sotto-classe rappresenta una variante specifica di tale concetto generale. Su questa interpretazione si basa tutta la teoria dell'ereditarietà nei linguaggi a oggetti. Oltre a essere un importante strumento di modellazione, l'ereditarietà ha importanti ricadute sulla riusabilità ed il design del software.

Quando una sotto-classe eredita da una super-classe, tutti gli attributi e i metodi di istanza della super-classe sono automaticamente presenti nella sotto-classe. Per questo motivo, il termine *ereditarietà*: la sotto-classe eredita almeno il comportamento (variabili e metodi) della super-classe. Proprio per

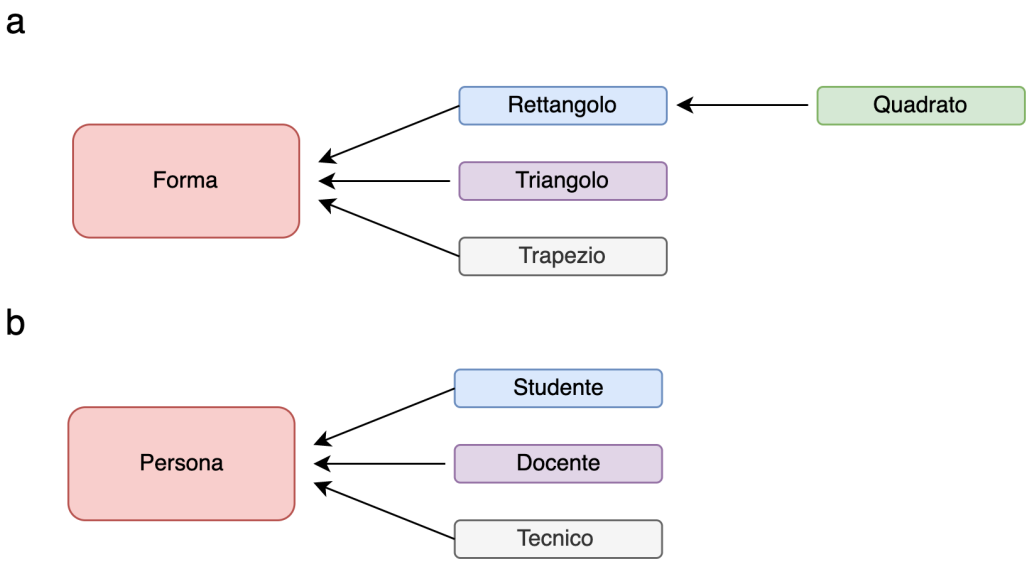


Figura 43. a) Esempio di relazione di ereditarietà. Un `Rettangolo`, un `Triangolo` ed un `Trapezio` sono sotto-classi di una generica classe `Forma` (super-classe). Inoltre, un `Quadrato` è una sotto-classe di un `Rettangolo` - perchè è un rettangolo con la particolarità di avere tutti i lati uguali. b) Con un simile ragionamento possiamo inutire che una ipotetica classe `Persona` sia la super-classe per classi specializzate quali uno `Studente`, un `Docente` o un `Tecnico`.

questo motivo la super-classe è una classe generale - almeno più generale della sotto-classe! - che serve a definire gli attributi comuni ad una serie di classi specializzate.

> **Esempio 14.2.** Supponiamo di voler modellare un gerarchia di concetti astratti quali gli "animali". Tutti i "mammiferi" sono animali, ma non tutti gli animali sono mammiferi. Quindi, se dovessimo modellare con delle classi gli oggetti mammifero e animale, allora sicuramente animale sarebbe la super-classe di mammifero. In maniera simile, potremmo pensare di modellare le forme geometriche, suddivise in quadrati, rettangoli, triangoli etc., oppure il personale di una università (Figura 43).

Continuando l'esempio della costruzione di un programma che modelli alcuni dei ruoli lavorativi esistenti all'interno dell'università. Chiediamoci, cosa accomuna tutte le persone che vorremmo descrivere? Senza alcuna presunzione, sicuramente tutti coloro che vogliamo descrivere sono delle persone ed hanno un'"anagrafica", che è indipendente dallo specifico ruolo ricoperto. Senza tergiversare tanto su cosa sia o meno opportuno come anagrafica - non è questo il punto - se definiamo il concetto di persona tramite una super-classe, allora ad esempio gli studenti, i docenti ed i tecnici universitari sono tutti esempi specializzati (sotto-classi) di una generica *persona*.

Cominciamo in codice a descrivere la super-classe:

```
# classe persona con anagrafica (super-classe)
class Persona:
    def __init__(self, ruolo, nome, cognome):
        self.ruolo = ruolo
        self.nome = nome
        self.cognome = cognome

    def bonjour(self):
        print(self.ruolo, ":", self.nome, self.cognome)
```

Esercizio 14.3. In questo caso, rispetto all'esempio precedente, abbiamo associato il `ruolo` al costrutture di `Persona`, perché?

A questo punto posso dichiarare una sotto-classe, la sintassi generale richiede di dichiarare

```
class nome_classe(nome_super_classe)
    .....
```

laddove ovviamente `nome_super_classe` rappresenta il nome della super-classe padre di `nome_classe`. Quindi possiamo definire una sotto-classe `Studente`, supponendo di voler memorizzare, oltre all'anagrafica che otteniamo da una generica `Persona`, anche ulteriori informazioni relative almeno ad un corso che uno studente potrebbe seguire (si veda l'esempio in Figura 44).

Riportiamo qui il codice di della sotto-classe `Studente`, il quale modella una persona per la quale si ha un corso associato.

```
# sotto-classe studente
class Studente(Persona):

    # costruttore sotto-classe
    def __init__(self, nome, cognome, corso):
        super().__init__("Studente UNITS", nome, cognome)
        self.corso = corso

    # ridefinizione del metodo bonjour di persona
    def bonjour(self):
        Persona.bonjour(self) # uso esplicitamente metodo di Persona
        print("> Frequento il corso: ", self.corso)
```

Analizziamo il costruttore della sotto-classe. Ogni studente è anche una persona, perciò i parametri del costruttore sono anche i parametri di un oggetto di classe `Persona`, ovvero `nome`, `cognome` ed un `corso`. Si noti invece che il `ruolo` non è un parametro del costruttore. A questo punto, per costruire uno `Studente` ci avvaliamo del costruttore della classe `Persona`, al quale possiamo passare esattamente i parametri necessari.

```
super().__init__("Studente UNITS", nome, cognome)
```

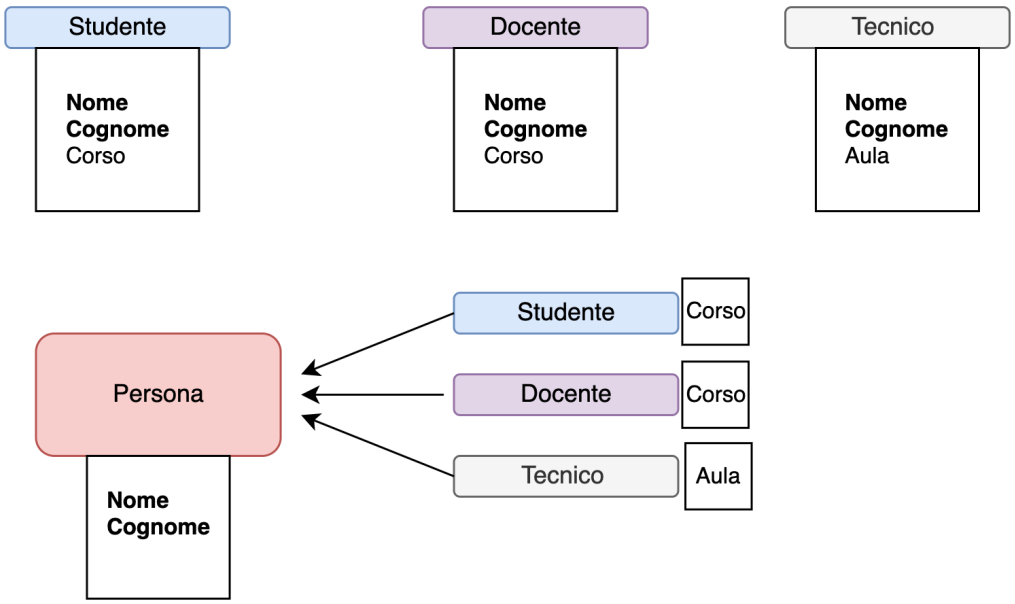


Figura 44. Possiamo automatizzare il processo che porta al punto b di Figura 43. Elenchiamo gli attributi che assegniamo a studenti, docenti e tecnici; supponiamo tutti abbiano una anagrafica (nome e cognome, in grassetto), più degli attributi specifici: il corso seguito da uno studente, il corso insegnato da un docente, oppure l'aula in cui il tecnico opera. A questo punto tutte le variabili di istanza condivise (nome e cognome), ha senso facciano parte del super-classe `Persona`, cosi che vengano definite una sola volta e siano disponibili in tutte le sotto-classi che estendono `Persona`. Gli altri valori rimangono invece privati alle sotto-classi perché non condivisibili (ad esempio, uno studente non viene associato ad una aula).

Successivamente si inizializzano gli attributi specifici di `Studente`. Si noti che per chiamare il costruttore della super-classe – ovvero `__init__` della classe `Persona`, non di `Studente`! – si usa la parola chiave `super()`. Come sarà intuitivo dal nome, `super` non è altro che una funzione che usiamo per specificare che vogliamo usare i metodi disponibili nella super-classe dell'oggetto `self`[7], e non quindi quelli di `Studente`.

> **Nota bene.** Se si chiama il costruttore della suoer-classe si devono passare tutti i parametri necessari. Si noti che qui, ad esempio, a `ruolo` associamo una costante `"Studente UNITS"`. Questo perchè nella nostra programmazione non vogliamo lasciare la libertà di cambiare questo valore di ruolo a chi costruisce gli oggetti della classe `Studente`.

Allo stesso modo si possono ridefinire metodi definiti nella super-classe, al cui interno viene chiamato il metodo omonimo della super-classe ed eventualmente vengono aggiunte funzionalità. Questo tipo di comportamento si chiama

[7] In pratica le classi e le loro gerarchie statiche sono memorizzate in una tabella delle classi. `super()` restituisce un riferimento all'entrata corretta di quella tabella, quella per `Persona`.

overriding - sovrascrittura - dei metodi della super-classe, e lo vediamo in pratica nella definizione di `bonjour`. Per fare overriding si deve ri-definire il metodo della super-classe esattamente con la stessa segnatura

```
def bonjour(self):
        ....
```

ma notate che posso decidere di chiamare il metodo della super-classe usando, ancora una volta una sorta di `super()`

```
        Persona.bonjour(self) # uso esplicitamente di un metodo di Persona
```

Un possibile utilizzo del precedente frammento di codice è:

```
# creo un oggetto e chiamo la funzione bonjour
obj_Giulio = Studente("Giulio", "Caravagna", "Programmazione")

obj_Giulio.bonjour()
```

Si verifichi che venga eseguita la funzione `bonjour` definita in `Studente`.

In modo analogo si può definire una classe `Docente`:

```
class Docente(Persona):
    def __init__(self, nome, cognome, corso):
        super().__init__("Docente UNITS", nome, cognome)
        self.corso = corso

    def bonjour(self):
        Persona.bonjour(self)
        print("> Docente del corso: ", self.corso)
```

Da notare che, per gli attributi che abbiamo definito per `Studente` e `Docente`, la super-classe `Persona` è la più generale che poteva esser definita, poiché entrambe le classi aggiungono solo un attributo a `Persona` (si veda l'esempio di Figura 44).

Possiamo istanziare un `Docente` nel seguente modo:

```
obj_Stefano = Docente("Stefano Alberto", "Russo", "Programmazione")
obj_Stefano.bonjour() # eseguita funzione della sotto-classe
```

Esercizio 14.4 (Modellazione a classi - seconda parte)**.** Con riferimento ai 5 esempi di modellazione a classi elencati nell'Esercizio 14.2, si ragioni ora della possibilità di introdurre delle gerarchie tra le classi elencate. Si discuta anche degli eventuali attributi e metodi che sarebbero di pertinenza delle eventuali super-classi, e delle sotto-classi. Per esempio, continuando l'esempio sullo zoo, gli animali potrebbero essere tutti sotto-classe della super-classe `Animale`, con variabili di istanza che memorizzano l'eta o altre caratteristiche comuni a tutti gli animali. I metodi della super-classe potrebbero fornire le operazioni per la gestione – ad esempio `nutrire`, `controllare_salute`, `spostare` etc. – ogni singolo animale dello zoo.

Esempio 14.3. Si vogliono modificare le classi `Studente` e `Docente` precedentemente definite in modo che esse possano:

- memorizzare una lista di corsi frequentata da uno studente;
- memorizzare una lista di corsi insegnati da un Docente;

Per semplicità le liste in questione non sono vuote. Si vuole quindi definire `bonjour` per stampare a schermo la lista completa dei corsi.

In pratica, si vuole permettere di scrivere il seguente codice:

```
corsi = ["Programmazione", "Laboratorio", "Analisi", "Geometria"]
obj_Giulio = Studente("Giulio", "Caravagna", corsi)
```

cosicché `obj_Giulio.bonjour()` stampi tutti i corsi frequentati da `obj_Giulio` (variabile `corsi`, passata al costruttre).

Una possibile soluzione è la seguente:

```
# sotto-classe Studente con una lista di corsi
# memorizzo anche n, il numero dei corsi seguiti
class Studente(Persona):
    def __init__(self, nome, cognome, lista_corsi)
        super().__init__("Studente UNITS", nome, cognome)
        self.lista_corsi = lista_corsi
        self.n = len(lista_corsi)

    def bonjour(self):
        Persona.bonjour(self)

        # stampo con un for
        for i in range(0, self.n):
            print("> Frequento il corso: ", self.lista_corsi[i])

# sotto-classe Docente con una lista di corsi
# memorizzo anche n, il numero dei corsi insegnati
class Docente(Persona):
    def __init__(self, nome, cognome, lista_corsi)
        super().__init__("Docente UNITS", nome, cognome)
        self.lista_corsi = lista_corsi
        self.n = len(lista_corsi)

    def bonjour(self):
        Persona.bonjour(self)

        # stampo con un for
        for i in range(0, self.n):
            print("> Insegno il corso: ", self.lista_corsi[i])
```

A questo punto lo studente è invitato a risolvere il seguente esercizio.

Esercizio 14.5 (Docenti mappati su studenti)**.** Si estendano ulteriormente le classi definite nell'Esempio 14.3 aggiungendo:

- un algoritmo che calcoli se un docente insegna tutti i corsi frequentati da uno studente.

- un algoritmo che verifichi che, per ogni studente iscritto ad un certo numero di corsi, esistano docenti che effettivamente insegnino quei corsi.

Per risolvere il secondo punto si usi la soluzione del primo.

Per un eccesso di bontà nella scrittura di questa dispensa riportiamo una possibile soluzione. Iniziamo con il definire una funzione ausiliaria - inutile per ora, ma non quando eseguirete il codice! - che stampi un separatore su console.

```
def separatore():
    print("--------")
```

A questo punto vogliamo scrivere una funzione `docente_copre_studente` che stampi a schermo se un docente insegna tutti i corsi frequentati da uno studente. Nell'ottica di risolvere anche il secondo punto dell'esercizio decidiamo di far si che la funzione restituisca l'eventuale lista di corsi scoperti per lo studente. In notazione insiemistica, se lo studente segue l'insieme dei corsi S, ed il docente insegna l'insieme dei corsi D, allora questa funzione restituisce

$$R = S \setminus D$$

definita mediante una sottrazione tra insiemi. R è l'insieme dei corsi non coperti da questo docente, per questo studente.

```
def docente_copre_studente(docente, studente):

    # Convenevoli
    print("Docente che esamino")
    separatore()
    docente.bonjour()

    print("\n Studente che esamino")
    separatore()
    studente.bonjour()

    print("\n Ricerca in corso...")
    separatore()

    # flag: True se tutti i corsi sono coperti
    corsi_trovati = True

    # lista di corsi scoperti, inizialmente vuota
    # In C questa sarebbe una lista linkata
    corsi_scoperti = []

    # cerco ogni corso dello studente
    for i in range(0, studente.n):

        # ipotizzo che il corso non sia stato trovato
```

```
        corso_trovato = False

        # vado a vedere tutti i corsi del docente
        j = 0

        while not corso_trovato and j<docente.n:
            if(studente.lista_corsi[i] == docente.lista_corsi[j]):
                corso_trovato = True
            j = j + 1

        # stampo l'info che dice se ho trovato o meno il corso
        print("Corso ", studente.lista_corsi[i], "->", corso_trovato)

        # se il corso non è stato trovato, appendo in fondo alla lista
        if (not corso_trovato):
            corsi_scoperti = corsi_scoperti + [studente.lista_corsi[i]]

        # Creo un and tra tutti i corsi
        corsi_trovati = corsi_trovati and corso_trovato

    # riporto a schermo se ho trovato tutto (fuori dal for)
    print("\n Ricerca completata")
    separatore()

    if(corsi_trovati):
        print("Tutti i corsi sono coperti")
    else:
        print("Qualche corso rimane scoperto")
        for i in range(0, len(corsi_scoperti)):
            print("\t >", corsi_scoperti[i])

    return corsi_scoperti
```

Commentiamo questa funzione:

- come schema di programmazione abbiamo usato la ricerca con flag vista nella Parte I di questa dispensa (Sezione 6.3); essendo un pattern di programmazione, anche se lo abbiamo visto in `C` lo possiamo usare in `Python`. Anticipiamo però che questo non sia un modo molto pythonico di scrivere questo algoritmo – dopo vedremo come mai!

- nel passaggio

  ```
  corsi_scoperti + [studente.lista_corsi[i]]
  ```

  appendiamo in fondo alla lista `corsi_scoperti` il nome del corso che non troviamo, `studente.lista_corsi[i]` – come ottenevamo con la `addC` nelle liste linkate in `C` (Sezione 8). Qui il `"+"` indica la concatenazione di due liste per la precisione; per poterlo usare possiamo costruire una lista attorno a `studente.lista_corsi[i]`, mediante le parentesi quadrate – una lista di un singolo elemento.

- la formula logica che costruisco è un "and" su ciascuno dei corsi dello studente. La variabile `corso_trovato` mi dice se ho trovato o no il corso i-esimo. Quindi `corsi_trovati` – booleano – è inizializzata a `True` altrimenti l'`and` sarebbe sempre `False`.

Adesso vogliamo scrivere una funzione che verifichi che, per ogni studente iscritto ad un certo numero di corsi, esistano docenti che effettivamente insegnino quei corsi. Ragioniamone da un punto di vista algoritmico, vista la funzione appena definita:

- supponiamo che uno studente debba seguire i corsi $S^{(1)}$ dove il (1) denota il passo di computazione - al primo passo $S^{(1)} = S$ (tutti i corsi sono ancora da coprire) ;
- prendiamo un docente – uno qualunque dei disponibili – che copre i suoi corsi $D^{(1)}$, laddove sperabilmente $D^{(1)} \cap S^{(1)} \neq \emptyset$;
- usiamo la funzione sopra definita per calcolare

$$R^{(1)} = S^{(1)} \setminus D^{(1)}$$

  ovvero definire l'insieme dei corsi che non sono coperti dal docente che insegna $D^{(1)}$ (ma magari dagli altri);
- procediamo ricorsivamente – nel senso di definire $S^{(2)} = R^{(1)}$ – a risolvere. A tal proposito possiamo immaginare di risolvere lo stesso problema supponendo di avere a disposizione un nuovo studente con corsi $S^{(2)}$, ed un pool di docenti uguale a quello iniziale, decurtato del docente che insegna $D^{(1)}$ (già testato).

Chiaramente questo schema di programmazione si basa su una sorta di relazione ricorsiva

$$S^{(i+1)} = S^{(i)} \setminus D^{(i)}$$

che si ferma quando o la lista di docenti da cui prendere un nuovo D è terminata, oppure quando $S^{(i+1)} = \emptyset$. In ambedue i casi, solo se alla terminazione non ho più corsi da coprire allora avremo copertura totale.

> **Nota bene.** Essendo uno studente un oggetto di una classe che noi abbiamo definito, possiamo davvero creare nuovi studenti istanziando la classe `Studente`! Quindi in pratica possiamo davvero creare un nuovo studente per modellare $S^{(i+1)}$.

Possiamo scrivere questa intuizione in Python come segue.

```
def copertura_totale(lista_di_docenti, studente):

    # ipotesi iniziale, non è coperto
```

```
coperto = False

# inizialmente devo verificare tutti i corsi
corsi_da_verificare = studente.lista_corsi

i = 0

print("Ricerca copertura")
separatore()

# iterazione indeterminata: appena tutti i corsi sono coperti esco dall'iterazione
while not coperto and i<len(lista_di_docenti):

    # creo un nuovo studente il cui attributo corsi
    # ha tutti i corsi che non sono stati coperti dai docenti esaminati fino ad ora
    obj_S_ast = Studente(studente.nome, studente.cognome, corsi_da_verificare)

    # calcolo i corsi che non sono coperti, oppure lista vuota
    copertura_docente = docente_copre_studente(lista_di_docenti[i], obj_S_ast)

    # se tutti i corsi sono stati coperti
    if(len(copertura_docente) == 0):
        coperto = True

    # se alcuni corsi ancora non sono coperti
    else:
        corsi_da_verificare = copertura_docente
    i = i + 1

print("Ricerca con tutti i docenti completata")
separatore()
```

Si invita lo studente di questa dispensa a testare il codice qui riportato.

> **Nota bene** (Variabili di istanza fuori dal costrutture)**.** In `Python`, a differenza di altri linguaggi, la struttura di oggetti è piuttosto "flessibile". Si consideri questo codice.
>
> ```
> class A:
>    def __init__(self, a):
>       self.a = a
>
>    def secondo(self, b):
>       self.b = b
> ```
>
> Qui `b` è una variabile di istanza degli oggetti di classe `A` a tutti gli effetti, solo che è disponibile solamente dopo aver chiamato il metodo `secondo`. In generale quindi lo stato interno di un oggetto può "aumentare" man mano che certi calcoli (i.e., funzioni) sono applicate agli stessi.

14.3 Classi astratte

Un ulteriore esempio - trovare le radici di una funzione ignota - ci mostra la potenza dell'ereditarietà e, allo stesso tempo, il concetto di computazione

astratta (detto diversamente, parzialmente specificata).

Data una funzione

$$f : [a, b] \to \mathbb{R}$$

vogliamo trovare il valore di $x_* \in [a, b]$ tale per cui $f(x_*)$ più vicino allo 0. In pratica, stiamo cercando

$$x_* = \min_{a \leq x \leq b} |f(x)| .$$

Questo è un problema generale, che può anche essere complesso, ma noi ci limitiamo ad affrontarlo dal punto di vista programmativo, limitando la sofisticatezza matematica per ragionare sulla programmazione.

Per risolverlo vogliamo implementare in `Python` un risolutore generico, ovvero che *prescinda dalla particolare forma della funzione* $f(\cdot)$ – f sarà solamente un funzione monadica, cioè di un singolo argomento x. La soluzione più facile è valutare la funzione f in certi punti calcolati discretizzando l'intervallo $[a, b]$ in n punti

$$x_1, x_2, \ldots x_n$$

e trovando tra questi il più vicino a 0. Qui possiamo banalmente definire

$$x_i = a + (i - 1)\delta_x$$

dove δ_x è un fattore di discretizzazione arbitrario. Inoltre, i è quindi l'indice dell'intervallo discreto di $[a, b]$, x_0 coincide con a, e $(b - a)/delta_x$ intervalli sono usati.

Questa soluzione richiede quindi che si riesca a calcolare il valore della funzione in punti arbitrari all'interno dell'intervallo $[a, b]$. Per risolvere questo problema sfruttando la programmazione ad oggetti, possiamo definire una classe per una f generica che, dato un particolare valore in input x, calcoli il valore della funzione in corrispondenza del punto in input: $f(x)$. Per fare questo ci troviamo ad introdurre la keyword `pass`.

```
class Function:
    def eval(self, x):
        pass
```

Si noti che il metodo `eval` in pratica non è implementato – non restituisce nulla. Ad ogni modo, definisce che ogni sotto-classe di `Function` avrà a disposizione questo metodo. Usando l'overriding visto prima, tale metodo potrà essere ri-definito nelle sotto-classi; successivamente approfondiremo questo aspetto dei metodi astratti.

> **Nota bene** (`pass`)**.** Il `pass` è in realtà una dichiarazione `null` che viene generalmente utilizzata come segnaposto. Quando si vuole dichiarare una funzione o un loop ma non si vuole fornire l'implementazione, allora si può usare la dichiarazione `pass`.

Ovvero questa classe rappresenta una funzione $f : \mathbb{R} \to \mathbb{R}$ che prenda in input un punto x del dominio e restituisca in output $f(x)$.

Proponiamo questa versione della generica funzione, implementando anche il calcolo del minimo come richiesto

```
class Function:

    # Costruttore, associo un nome mnemonico alla funzione,
    # e gli estremi dell'intervallo
    def __init__(self, name, a, b):
        # nome funzione
        self.name = name

        # intervallo di riferimento
        self.a = a
        self.b = b

    # Funzioni ausiliarie: stampa nome
    def printname(self):
        print("Function: ", self.name, " tra [", self.a, ";", self.b, "]")

    # Funzioni ausiliarie: stampa intervallo di valori
    def prettyprint(self, message, values):
        drop = False

        if(type(values) is list and len(values) > 10):
            drop = True
            values = values[1:10]
            values = values + ["..."]

        if(not drop):
            print("------")
        else:
            print("------ (mostro solo i primi 10 valori) ")
            print(message, "->", values)
            print("------\n")

    # metodo che risolve il problema algebrico descritto sopra
    def min_value(self, delta_x = 0.5):

        print("Calcolo valore più vicino a 0")
        self.printname()

        # range di calcolo da a a b, con step delta_x
        x_start = self.a
        x_end = self.b
        i = 0

        # f_x memorizza f(x)
        f_x = []

        # abs_f_x memorizza |f(x)|
```

```
        abs_f_x = []
        x = []

        # valuto la funzione in punti all'interno di [a, b]
        # che distano tra loro delta_x
        while(x_start + i*delta_x) < x_end:
            this_x = x_start + i*delta_x
            # calcolo quanto vale la funzione nel punto di ascissa this_x
            this_fx = self.eval(this_x)
            x = x + [this_x]
            f_x = f_x + [this_fx]
            abs_f_x = abs_f_x + [abs(this_f_x)]

            i = i + 1

        self.prittyprint("Dominio x ", x)
        self.prittyprint("Funzione f(x) ", f_x)
        self.prittyprint("Valore assoluto |f(x)| ", abs_f_x)

    # per cercare il valore più vicino a 0 guardo i valori assoluti
    # all'indice in cui trovo il minimo ho trovato il valore che è più vicino a 0
    # assumo che la posizione 0 sia il minimo inizialmente
    x_min = 0
    for i in range(1, len(abs_f_x)):
        # se trovo un valore più piccolo del minimo corrente
        if abs_f_x[i] < abs_f_x[x_min]:
            # l'indice del minimo è i
            x_min = i

    print("L'elemento più vicino a 0 x = ", x[x_min], " con f(x) = ",
    self.eval(x[x_min]]))

    # Metodo di valutazione - pass!
    def eval(self, x):
        pass
```

Si noti che ancora una volta abbiamo usato uno schema di programmazione alla `C` del tutto efficace, nonostante anche questa parte potesse essere resa molto più pythonica (vedasi dopo).

Ad ogni modo, la cosa importante è che `Function` implementa il nostro algoritmo in tutto e per tutto. Unica cosa, non lo può davvero eseguire perchê linee come

```
this_fx = self.eval(this_x)
```

oppure

```
print("L'elemento più vicino a 0 x = ", x[x_min], " con f(x) = ",
    self.eval(x[x_min]]))
```

sono indefinite (`pass`).

Possiamo adesso creare delle sotto-classi che ereditino dalla classe `Function` e che descrivano funzioni di una forma specifica, ad esempio una retta con equazione

$$f(x) = mx + q$$

```
# sotto-classe che descrive una retta
class Retta(Function):
    def __init__(self, a, b, m, q):
        # inizializzo la super-classe (funzione su intervallo)
        super().__init__("Retta", a, b)

        # aggiungo come attributi coefficiente angolare e intercetta
        self.m = m
        self.q = q

    def printname(self):
        # chiamo il metodo omonimo della classe madre
        Function.printname(self)

        # aggiungo funzionalità
        print("Coeff angolare m = ", self.m, " e intercetta q = ", self.q)

    # implemento eval sfruttando la forma funzionale della retta
    # ovvero f(x) = m*x + q
    def eval(self, x):
        return self.m * x + self.q
```

Adesso posso semplicemente testare la mia classe; lo studente provi ad eseguire il seguente codice e testi analiticamente la bontà del risultato.

```
# esempio di utilizzo
bisettrice = Retta(-3, 3, 1, 0)
bisettrice.min_value(delta_x = 1)
```

Questa situazione implementativa sembra particolarmente versatile. Per realizzare altre "funzioni" mi basta quindi di estendere `Function`. Ad esempio una parabola con equazione

$$f(x) = c_2x^2 + c_1x + c_0$$

```
# sotto-classe che descrive una parabola
class Parabola(Function):
    def __init__(self, a, b, c0, c1, c2):
        # inizializzo classe madre
        super().__init__("Parabola", a, b)
        # aggiungo come attributi i tre coefficienti
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def printname(self):
        Function.printname(self)
```

```
        print("Parabola con coefficienti a = ", self.c0, "b = ", self.c1, "c = ", self.c2, "\n")

    # implemento eval sfruttando la forma funzionale della parabola
    # ovvero f(x) = c_0 + c_1*x + c_2*x*x
    def eval(self, x):
        return self.c0 + self.c1 * x + self.c2 * (x*x)
```

A cui corrisponde un semplice esempio di utilizzo.

```
parabola_semplice = Parabola(-3, 3, 0, 0, 1)
parabola_semplice.min_value(delta_x = 1)
```

Si invita lo studente di questa dispensa a provare ad eseguire il codice `Python` qui riportato, e confrontarlo con una soluzione analitica.

> **Nota bene.** Ci sono altri modi per ottenere lo stesso comportamento. Ad esempio, si esamini e commenti la seguente gerarchia di classi.
>
> ```
> class Function:
>
>     def eval(self, x):
>         pass
>
> class IntervalFunction(Function):
>     # i parametri sono gli estremi dell'intervallo su cui lavoriamo
>     def __init__(self, a, b):
>         self.a = a
>         self.b = b
>
>     # override del metodo eval
>     def eval(self, x):
>         if(x<a or x>b):
>             # sollevo eccezione
>             raise Exception("Fuori intervallo!")
>         else:
>             innereval(self, x)
>
>     # eval su punti interni dell'intervallo
>     def innereval(self, x):
>         pass
> ```
>
> Ovvero specifico che voglio lavorare all'interno dell'intervallo $[a, b]$ e sollevo un'eccezione se il punto x passato come argomento alla funzione `eval` è fuori dell'intervallo, oppure chiamo una funzione ausiliaria `innereval` se il punto x è correttamente all'interno dell'intervallo.

14.3.1 Il decoratore abstractmethod

CIRO: Breve excursus sui decoratori, spiegare che cosi come detto sopra non abbiamo obbligato nessuno a definire eval, descrizione di abstractmethod come modo di vincolare.

15 Meccanismi avanzati di iterazione

15.1 List comprehensions

In alcuni degli esempi delle Sezioni 14.1, 14.2 e 14.3 abbiamo esplicitamente criticato lo stile di programmazione adottato, giudicandolo "troppo vicino" a quello visto in `C` nella prima parte di questa dispensa. Infatti, in `Python`, abbiamo alcuni meccanismi di programmazione che permettono di ottenere lo stesso risultato del codice mostrato precedentemente, con uno sforzo molto minore. Essere un buon programmatore `Python` richiede quindi di conoscere e sfruttare tali meccanismi.

In questa sezione introduciamo uno di questi meccanismi: la **list comprehension**, un modo semplice per creare liste e sottoliste in `Python`. La comprehension rende i nostri programmi estremamente più facili da leggere, e rapidi da scrivere. Per introdurla, faremo un confronto con la modalità "classica" di creazione di liste, quella che abbiamo usato negli esercizi precedenti.

Consideriamo che una lista è formata da elementi provenienti da altre sequenze (e.g., i numeri da 1 a 10) su cui sono state svolte determinate operazioni, o da sotto-sequenze che rispettano determinate condizioni (e.g., i numeri pari da 1 a 10). Ad esempio, supponiamo di aver bisogno di una lista di quadrati a partire da un elenco di numeri.

In modalità `C`, procederemmo come segue:

1. creiamo una lista vuota `lista`;
2. aggiungiamo a `lista` i quadrati dei numeri da elevare, il tutto utilizzando un ciclo `for`;

In `Python`, usando la `append` – che è esattamente la funzione `addC` della Sezione 8 in `C` – possiamo scrivere

```
quadrati = []

for n in range(10):
    quadrati.append(n**2)
```

da cui la stampa attesa

```
print(quadrati) #[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

La list comprehension per questa lista è invece

```
# Una linea
quadrati = [n**2 for n in range(10)]

print(quadrati) #[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

il cui effetto è evidentemente lo stesso – la stessa lista di quadrati viene creata, ma con solo una riga di codice!

La sintassi "astratta" – nella versione più samplice – di una list comprehension segue questo pattern

$$\texttt{[} f(\texttt{x}) \texttt{ for x in } X \texttt{]}$$

cosí da creare la lista

$$\texttt{[} f(\texttt{x}_1),\ f(\texttt{x}_2),\ \ldots,\ f(\texttt{x}_n) \texttt{]}$$

laddove $X = \{\texttt{x}_1, \texttt{x}_2, \ldots, \texttt{x}_n\}$. Si noti che è semplice leggere la definizione stessa della lista per capire il significato della comprehension: in questo caso sono tutti e soli gli elementi $\texttt{x} \in X$, ai quali applico f.

> **Nota bene.** Si noti come la comprehension sia formalmente scrivibile con un quantificatore universale ($\forall$) come quelli discussi in Sezione 6.3

Nell'esempio precedente X sarebbe `range(10)`, ovvero i numeri da 0 a 9, e la funzione f sarebbe il quadrato. Notate che non si definisce una funzione vera e propria, infatti `n**2` è semplicemente una espressione il cui scoping è legato al lato destro della comprehension – quindi a `x in` X.

Esercizio 15.1. Si crei una lista identica a quella di partenza mediante comprehension.

```
x = [1,2,3,4,5]

y = .... # comprehension.

print(x) # [1,2,3,4,5]
print(y) # [1,2,3,4,5] ?
```

Vediamo un altro esempio semplice per complicare un poco la comprehension: vogliamo combinare i valori presenti in due liste, purché questi siano diversi tra loro.

```
lista_uno = [1, 2, 3]
lista_due = [3, 1, 4]

mix = []

# annidamento di cicli for
for x in lista_uno:
    for y in lista_due:
        if x != y:
            mix.append((x, y))

print(mix) # [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Oppure, in una singola riga con una comprehension

```
mix = [(x, y) for x in lista_uno for y in lista_due if x != y]

print(mix) #[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Notate che qui usiamo una doppia comprehension, nel senso di quantificare su due insiemi di oggetti. Inoltre, qui utilizziamo anche un predicato per descrivere una appartenenza condizionata – solo se il predicato è vero. La forma generale di comprehension a singola iterazione sarebbe

$$\texttt{[} f(\texttt{x}) \texttt{ for x in } X \texttt{ if p(x)]} = \texttt{[} f(\texttt{x}_{i_1}), f(\texttt{x}_{i_1}), \ldots, f(\texttt{x}_{i_n}) \texttt{]}$$

laddove $P_X = \{\texttt{x}_{i_1}, \texttt{x}_{i_2}, \ldots, \texttt{x}_{i_n}\} \subseteq X$ è il sottoinsieme

$$P_X = \{\texttt{x}_i \mid \texttt{x}_i \in X \wedge p(\texttt{x}_i)\}\,.$$

Esercizio 15.2. Si prendano i codici `Python` degli esempi mostrati fino ad ora, in particolare quello riguardante la copertura dei corsi (Esempio 14.3), e la funzione `def min_value(self, delta_x = 0.5)` nell'esempio riguardante lo studio di funzione (classe `Function`), e li si riscriva mediante list comprehension laddove possibile (e sensato).

15.2 Iteratori

In una comprehension scriviamo

```
[.... for x in X]
```

Ora, tutti noi concordiamo nell'intuizione di dire "per ciascun elemento x di X", ma un computer che interpreta un programma non può basarsi sull'intuizione. Lo studente attento avrà a questo punto notato che ogni cosa "intuitiva" deve in realtà basarsi su qualcosa di rigoroso, generale e, soprattutto, chiaro.

In questa sezione, introduciamo il concetto di **iteratore**, il modo per navigare (o camminare, o generare, etc.) i valori che sono memorizzati all'interno di una collezione appunto "iterabile".

> **Nota bene** (Collezioni iterabili)**.** Una collezione iterabile è un modo di rappresentare gli oggetti secondo una qualche struttura, definendo il concetto di cammino sugli stessi. Per esempio, se pensiamo ad i numeri da 1 a 10, il cammino più semplice è quello che genera
>
> $$1, 2, 3, \ldots, 10$$
>
> In alternativa, se penso ad una stringa `"banana"`, il cammino più semplice potrebbe essere quello che scandisce le lettere di `"banana"` una per ciascuna
>
> $$\texttt{"b"}, \texttt{"a"}, \texttt{"n"}, \ldots, \texttt{"a"}$$
>
> Adesso, si noti bene che il modo in cui si decide di fare un determinato cammino è completamente arbitrario. Potrei, per motivi miei, decidere di saltare a caso fra i numeri da 1 a 10, oppure saltare solo tra da un pari

all'altro! L'importante, è che se una collezione è iterabile, allora posso generare dei cammini su di essa.

In `Python` un iteratore è un oggetto su cui è possibile iterare, ovvero un oggetto che ha una nozione *implicita* di iterazione. Il meccanismo di iterazione ritorna un elemento per volta e viene realizzato implementando necessariamente due magic methods:

- `__iter__`, che inizializza l'iteratore e lo ritorna tramite `self` (perchè l'iteratore essendo implicito fa parte dell'oggetto stesso, quindi è `self`);
- `__next__`, che ritorna il prossimo valore dell'iteratore da visitare.

Cominciamo con vedere il semplice utilizzo di un iteratore

```
mystr = "banana"
myiter = iter(mystr) # costruisco iteratore

print(next(myiter)) # b
print(next(myiter)) # a
print(next(myiter)) # n
print(next(myiter)) # a
print(next(myiter)) # n
print(next(myiter)) # a
```

In questo caso costruisco l'iteratore con la chiamata `iter(mystr)`, usando il fatto che qualcuno ha implementato un magic method `__iter__` per le stringhe in `Python`. Le chiamate che restituiscono i valori su cui si muove l'iteratore - in questo caso i caratteri della stringa - sono tutte della forma `next(myiter)`.

Esempio 15.1 (Iteratore indefinito)**.** Costruiamo un semplicissimo iteratore che generi i numeri in modo indefinito, i.e.,

$$1, 2, 3, 4, \ldots$$

Per farlo, l'iteratore l'unica cosa che deve ricordarsi è l'ultimo valore, chiamiamolo a, che ha restituito nelle chiamate a `next`. Per ottenere questo effetto di "memoria" basta utilizzare le variabili di istanza dell'oggetto che stiamo costruendo – aggiungiamo una variabile `a` a `self`!

```
class MyNumbers:

    # Costruzione dell'iteratore 1, 2, ...
    def __iter__(self):
        self.a = 1 # self.a viene inizializzato ad 1
        # iter deve ritornare l'oggetto stesso
        return self

    def __next__(self):
        x = self.a # copia di self.a in x
        self.a += 1 # incremento self.a di 1
        return x # restituisco x
```

Avendo una concezione ad oggetti per gli iteratori, posso facilmente testare la nostra implementazione instanziando un oggetto di classe `MyNumbers`.

```
myclass = MyNumbers()
myiter = iter(myclass) # costruisco iteratore

print(next(myiter)) # 1
print(next(myiter)) # 2
print(next(myiter)) # 3
print(next(myiter)) # 4
```

Ad ogni modo, `mystr` ha un numero finito di caratteri, che cosa succede se chiamo un'altra volta `next(myiter)`?

```
print(next(i)) # Eccezione: StopIteration!
```

Viene sollevata quindi una eccezione di tipo `StopIteration`; si veda la dispensa di Laboratorio di Programmazione per un'introduzione alla gestione delle eccezioni in `Python`. Per fermare una iterazione dobbiamo sollevare una eccezione.

Esempio 15.2 (Iterazione definita)**.** Prendiamo l'esempio precedente, ora vogliamo però definire un iteratore che generi tutti i valori interi dell'intervallo $[a, b]$, perciò

$$a, a+1, a+2, \ldots, b$$

Ovviamente, gli estremi a e b dell'intervallo possono essere passati a tempo di creazione dell'oggetto su cui costruirò l'iteratore mediante due parametri `low` e `high`.

```
class Series:

    # Costruttore, definisce a e b
    def __init__(self, low, high):
        # variabile locale che dice a quale punto sono della serie
        self.current = low
        self.high = high # costante

    # Se ho gia settato current a low, posso non fare nulla
    def __iter__(self):
        return self

    def __next__(self):
        # se sono fuori dell'intervallo di riferimento
        if self.current > self.high:
            # sollevo un'eccezione
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

La cosa confortevole di un iteratore è che, dopo averlo costruito, posso al-

leggerire la scrittura del mio codice sfruttandolo. Per esempio, riferendoci all'Esempio 15.2 in questo caso possiamo ciclare esplicitamente sugli elementi della nostra serie usando un ciclo `for`.

```
# 1, 2, 3, ... , 10
myS = Series(1, 10)

# ciclo esplicitamente sulla classe
for s in myS:
    print(s)
```

In maniera analoga, possiamo usare una list comprehension al posto del `for`.

```
[print(s) for s in Series(1, 10)]
```

E, senza stupirci, possiamo costruire una lista `Python` a partire dai nostri valori iterabili.

```
# costruisco una lista a partire da un oggetto di tipo Series
n_list = Series(1, 10)
print(list(n_list))
```

> **Nota bene** (Interruzione di iterazione)**.** Si osservi come sollevare esplicitamente l'eccezione `StopIteration` serva a interrompere l'iterazione, che altrimenti non si fermerebbe mai!

Il pattern generale per scorrere il nostro iteratore è quindi:

```
# inizio l'iterazione
for element in iterable:
    # faccio qualcosa con element
```

Questo pattern corrisponde in realtà al seguente codice.

```
# creo iteratore da oggetto iterabile
iter_obj = iter(iterable)

# loop infinito
while True:
    try:
        # prendi il successivo elemento
        element = next(iter_obj)

        # faccio qualcosa con element

    # in caso di StopIteration, esci dal loop
    except StopIteration:
        break
```

Esercizio 15.3. Come mai la "semantica" dell'iteratore viene implementata con un costrutto `while`?

Possiamo utilizzare il concetto di iteratore per definire un flusso di dati che magari non sono stati pre-calcolati. Ad esempio, le potenze di un certo numero.

Esempio 15.3. Vogliamo creare un iteratore che stampi tutte le potenze di 2, fino a che non raggiunge un esponente massimo x, ovvero

$$2^0, 2^1, 2^2, \dots 2^x$$

```
class PowTwo:
    # ricordo x
    def __init__(self, x):
        self.max = x

    # mi preparo ad iterare da 2^0=1
    def __iter__(self):
        # esponente corrente
        self.n = 0 # inizio da 2^0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2**self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

A questo punto la class `PowTwo` può essere utilizzata per generare il flusso dei dati previsti.

```
numbers = PowTwo(3)
i = iter(numbers)

print(next(i)) # 2^0 = 1
print(next(i)) # 2^1 = 2
print(next(i)) # 2^2 = 4
print(next(i)) # 2^3 = 8
print(next(i)) # Eccezione: StopIteration
```

Esempio 15.4 (Successione di Fibonacci)**.** Si vuole costruire un iteratore che generi i termini F_i della successione di Fibonacci, limitandolo a calcolare F_M con M un massimo scelto dall'utente.

```
class MyFibo:

    def __init__(self, max):
        # massimo indice dei termini generati
        self.max = max

    def __iter__(self):
        self.n = 0 # elemento indice 0 della successione
        self.n_0 = 0 # F_0
        self.n_1 = 1 # F_1
        return self
```

```
    def __next__(self):
        # controllo se sono all'interno dei valori che devo calcolare
        if self.n <= self.max:
            if self.n == 0:
                # ho memorizzato il valore
                result = self.n_0
            if self.n == 1:
                # ho memorizzato il valore
                result = self.n_1
            if self.n>1:
                # termine n-esimo F_n
                result = self.n_0 + self.n_1
                # memorizzo F_{n-1}
                self.n_0 = self.n_1
                # memorizzo F_n
                self.n_1 = result
            # incremento n
            self.n += 1
            return result
        else:
            raise StopIteration

# esempio di utilizzo
fibo = MyFibo(20)
for i in fibo:
    print(i)
```

Riprendendo le classi viste negli esempi precedenti riguardanti studenti e docenti (Sezione 14.1), supponiamo di voler modellare una università intesa come collezione di studenti e docenti. In particolare, vogliamo implementare un iteratore nella classe `Università` che permetta di eseguire qualche operazione per ogni `Studente` iscritto all'università.

```
class Universita:

    # Memorizzo le liste nel costruttore
    def __init__(self, lista_studenti, lista_docenti):
        self.lista_studenti = lista_studenti
        self.lista_docenti = lista_docenti

    # Inizialmente non ho ancora visto nulla
    def __iter__(self):
        self.indice = -1
        return self

    def __next__(self):
        # prima incremento indice dato che parto da -1
        self.indice += 1
        if(self.indice == len(self.lista_studenti)):
            raise StopIteration
        return self.lista_studenti[self.indice]
```

Grazie a questo semplice iteratore è possibile scrivere codice estremamente leggibile e versatile, sfruttando ad esempio una list comprehension.

```
# Creo una università
universita_units = Universita([obj_Giulio, obj_Mario], [obj_ProfRusso, obj_ProfPippo])

# Faccio qualcosa per ciascuno studente
[studente.bonjour() for studente in universita_units]
```

15.3 Liste concatenate in `Python`

Concludiamo questa dispensa costruendo, in `Python`, una struttura dati analoga alla lista linkata che abbiamo implementato in `C`.

```
class ListaConcatenata:
    # inner class: visibile solo all'interno della classe ListaConcatenata
    # analogo a ElementoDiLista in C
    class Nodo:
        val = None # valore interno al nodo
        prossimo = None # elemento next della struct in C

    # costruttore
    # stessa funzione del puntatore ListaDiElementi in C
    def __init__(self):
        self.testa = None # puntatore alla testa della lista

    # funzione che aggiunge un elemento alla lista
    # stessa funzionalità di addC in C
    def aggiungi(self, n_val):
        # se la lista è vuota
        if self.testa = None:
            # costruisco un nodo
            n_nodo = self.Nodo()
            # gli assegno il valore in input
            n_nodo.val = n_val
            # assegno il valore di testa
            self.testa = n_nodo
        # se la lista contiene già almeno un elemento
        else:
            # copio la testa, poiché non voglio modificare self.testa
            elem = self.testa
            # quando elem.prossimo == None sono all'ultimo elemento della lista
            while elem.prossimo != None:
                # sposto elem a elem.prossimo
                elem = elem.prossimo
                # creo un nuovo nodo
                n_nodo = self.Nodo()
                n_nodo.val = n_val
                elem.prossimo = n_nodo

# esempio di utilizzo
```

> **Nota bene.** Dal momento che `Python` non è un linguaggio fortemente tipato, il campo `val` può contenere oggetti di qualunque tipo.

E' naturale pensare di implementare un iteratore sulle liste, ovvero posso aggiungere alla classe definita le seguenti funzioni:

```
# ... all'interno della classe ListaConcatenata ...
def __iter__(self):
    # itero a partire dall'inizio della lista
    self.corrente = self.testa
    return self

def __next__(self):
    # se ho raggiunto la fine della lista
    if self.corrente is None:
        raise StopIteration
    # se ci sono ancora elementi da visitare
    val_corrente = self.corrente.val # valore oggetto corrente
    # mi sposto al prossimo elemento della lista
    self.corrente = self.corrente.prossimo
    return val_corrente
```

Con queste liste fatte in casa possiamo fare le seguenti operazioni.

```
lista_con = ListaConcatenata()

# Aggiungo elementi
lista_con.aggiungi(3)
lista_con.aggiungi(4)

# Posso usare gli stessi pattern di programmazione visti sulle liste in C
# ad esempio per stampare tutti gli elementi della lista:
corrente = lista_conc.testa
while corrente is not None:
    print(corrente.val)
    corrente = corrente.prossimo

# Oppure posso sfruttare implicitamente l'iteratore creato
for elem in lista_conc:
    print(elem)
```

PART

IV

Esercizi

16 Ereditarietà

Esercizio 16.1. Data la classe

```
class Veicolo:

    def __init__(self, modello, velocita, km):
        self.modello = modello
        self.velocita = velocita
        self.km = km
```

Si crei una classe `Autobus` che contenga, oltre alle informazioni su un veicolo, la capienza massima e le informazioni sulla rotta seguita (tramite una lista). Si forniscano dei metodi di stampa - nella super-classe o nella sotto-classe? - che stampano informazione su un veicolo generico, e su un autobus.

Esercizio 16.2. Con riferimento all'esercizio precedente, si aggiunga la classe `Auto` (oltre ad `Autobus`) con un metodo di istanza `revisionata` per controllare se la revisione sia stata fatta, o meno. Per verificare ciò `revisionata` prende in input l' anno attuale (`anno`), e testa se il veicolo è coperto per l'`anno` in input. A tal proposito, si devono chiaramente aggiungere eventuali informazioni riguardanti la revisione tramite variabili interne; si stabilisca vantaggi e svantaggi di metterli nella super-classe o nella sotto-classe, e si implementi l'approccio ritenuto migliore.

Esercizio 16.3. Con riferimento all'esercizio precedente, vogliamo fare le revisioni anche agli oggetti di classe `Autobus`. Si discuta se, con la soluzione svolta per l'esercizio precedente, l'aggiunta ora richiesta richiede tanto o poco aggiornamento di codice. Si risolva nuovamente questo ed il precedente esercizio per ridurre al minimo il codice necessario per avere le revisioni disponibili negli autobus.

Esercizio 16.4. Si crei una sotto-classe `Autobus` della super-class `Veicolo`. Il costo di default per una corsa completa su di un veicolo è 100 volte la capacità (in posti) del veicolo. Se il veicolo è un bus, c'è una maggiorazione del 10% sul costo di base (e.g., se la capacità del bus è di 50 posti, il costo dovrà essere 5500). Si usi il seguente codice.

```
class Veicolo:
    def __init__(self, modello, km, posti):
        self.modello = modello
        self.km = km
        self.posti = posti
```

```
    def fare(self):
        return self.posti * 100

class Bus(Veicolo):
    pass

scuolabus = Bus("Volvo", 12, 50)
print("Totale Bus:", scuolabus.fare())
```

Si noti che per implementare il calcolo si richiede l'overriding del metodo `fare()`.

Esercizio 16.5 (Esponenti e potenza)**.** Si scriva una classe potenza per definire le potenze di numeri con esponente sia positivo che negativo.

Esercizio 16.6 (Insieme potenza)**.** Si consideri il seguente codice

```
class pset:
    def sub_sets(self, sset):
        return self.subsetsRecur([], sorted(sset))

    def subsetsRecur(self, current, sset):
        if sset:
            return self.subsetsRecur(current, sset[1:]) + self.subsetsRecur(current + [sset[0]], sset[1:])
        return [current]
```

Si spieghi che tipo di calcolo viene effettuato da questa classe.

> **Nota bene.** Le slice di liste sono introdotte nella dispensa del corso di Laboratorio di Programmazione!

Esercizio 16.7 (Compagnia di viaggi)**.** La compagnia di viaggio "PT - Partire per non Tornare" ha ingaggiato il vostro team per realizzare un programma in grado di gestire i viaggi ed i clienti: i viaggi hanno molte caratteristiche in comune e qualche particolarità che le contraddistingue in base alla tipologia di viaggio (Mare, Montagna, Invernale o Estiva).

Attributi comuni:

- nome viaggio, identifica il titolo (a tema) di una vacanza;
- data partenza, identifica la data di inizio della vacanza e può essere rappresentato come una lista di tre elementi (giorno, mese, anno);
- data ritorno, identifica la data di conclusione della vacanza, anche essa può essere rappresentata come una lista di tre elementi;
- località, identifica la destinazione per la vacanza;
- resort, identifica l'albergo o il complesso turistico dove alloggeranno i clienti;
- prezzo, identifica il prezzo, a persona, per poter partecipare alla vacanza;

- partecipanti, identifica l'elenco dei clienti che si sono prenotati per il viaggio, e possono essere rappresentati attraverso una lista di cognomi;
- responsabile viaggio, identifica un dipendente della compagnia di viaggi, referente del viaggio.

Vacanza Invernale

- skipass, identifica il prezzo di uno skipass giornaliero;
- impianti sciistici, identifica l'elenco degli impianti sciistici convenzionati con l'agenzia di viaggio, rappresentabile come una lista;

Vacanza Estiva

- distanza, identifica la distanza, in metri, dal Resort alla spiaggia;
- escursioni marine, identifica l'elenco delle attività marittime a cui i clienti possono partecipare gratuitamente (snorkeling, parapendio, lezioni di surf, etc.).

La PT vorrebbe, inoltre, che da un viaggio fosse possibile determinare alcune utili informazioni.

Metodi Viaggio:

- `stampa`, funzione per stampare su schermo tutte le informazioni associate ad un determinato viaggio;
- `periodo`, funzione per determinare quanti giorni o mesi di vacanza sono stati programmati, come differenza della data di partenza e quella di ritorno;
- `guadagno`, funzione per determinare quale sia il ricavo netto dal viaggio, considerato che il 47% del prezzo di una prenotazione viene usato per coprire le spese del viaggio.

Si adattino le precedenti funzioni per i casi distinti di vacanza invernale e vacanze estiva. In particolare:

- il prezzo dello skipass pesa sulle spese del singolo ospite un ulteriore 15%;
- il prezzo delle escursioni marine pesa sulle spese del singolo ospite un ulteriore 10%.

Si realizzino le classi e sotto-classi necessarie per soddisfare le richieste della compagnia di viaggio. Inoltre, realizzare una funzione esterna che, dato il cognome di un preciso cliente ed una lista di tutti i viaggi organizzati dalla PT, determina i viaggi a cui il cliente ha partecipato e l'ammontare di denaro speso dal cliente per questi viaggi (la funzione non ritorna nulla).

Esercizio 16.8 (Ornitorinchi)**.** Vogliamo fare esperimenti sugli ornitorinchi, detti anche platipi. Durante la stagione riproduttiva, ogni platipo depone un certo numero di uova. Vogliamo costruire un modello per tenere traccia di quante uova ha deposto un platipo in ogni stagione.

- implementare un classe che prende in input il nome del platipo e una lista contenente il numero di uova deposte in ogni stagione. Per esempio

```
Perry = Platipo('Perry', [3, 2, 4, 1, 2])
```

- Espandere la classe Platipo con i seguenti tre metodi:
  - `Uova_totali`, che ritorna il numero totali di uova deposte;
  - `Stagioni_feconde`, che ritorna il numero di stagioni riproduttive in cui il platipo ha deposto almeno un uovo;
  - `Deponi_uova` che aggiunge le uova deposte alla lista di quelle già deposte.

Sappiamo che i platipi possono deporre fino a 3 uova ogni stagione. Partendo dall'assunzione che ogni numero di uova ha la stessa probabilità di essere deposta (incluso zero), vogliamo modellare la dinamica riproduttiva di un gruppo di 3 platipi.

Creare 3 oggetti platipi, scegliendo un nome e un numero di uova deposte in una stagione (i.e. una lista di un elemento).

Simulare 10 stagioni riproduttive, per ogni stagione generare per ogni platipo un numero randomico tra 0 e 3 di uova deposte, ed aggiungerlo alla lista del platipo.

Una volta che la simulazione è terminata stampare il nome, uova totali e stagione feconde per ogni platipo con la seguente sintassi: `"Billy ha depositato un totale di 10 uova, in 5 stagioni riproduttive feconde"`

> **Nota bene.** I numeri randomici possono essere generati con la funzione `randint(a,b)` presente nella libreria `random`; `a` e `b` sono l'estremo inferiore e superiore.

17 Classi astratte

Esercizio 17.1 (Polinomi)**.** Si consideri una la definizione generale di polinomio grado n

$$f(x) = a_1 x^n + a_2 x^{n-1} + \dots a_n x + a_{n+1}$$

- si definisca la classe più generale possibile per descrivere tali polinomi, e si descrivano di conseguenza come sotto-classi i polinomi di grado $n = 1$, $n = 2$ ed $n = 3$;

- si implementi, nella super-classe, un algoritmo per calcolare il massimo del polinomio in un intervallo $[a, b]$;
- si crei un test dove si creano diversi polinomi per ciascuno dei gradi scelti, e si stabilisca ogni volta quale sia il massimo.
- nel test, identificare una parametrizzazione per cui il massimo assoluto – massimo tra tutti i polinomi del test – sia quello del polinomio di grado $n = 1$.

Esercizio 17.2 (Retta valore assoluto)**.** Si consideri una funzione

$$f(x) = mx + q$$

ed il suo equivalente $f_{\|}(x) = |f(x)|$. Ci vengono date 3 classi:

- `Function` con un metodo astratto `eval`;
- `AbsoluteValueFunction` con un `eval` concreto per calcolare $f_{\|}$;
- `Retta` con un `eval` concreto per calcolare f.

Quale gerarchia deve essere in gioco tra queste classi per avere una corretta implementazione di f ed $f_{\|}$?

Esercizio 17.3 (Media di funzione)**.** Vogliamo calcolare il valore medio di una funzione definita su un intervallo $[a, b]$, usando la seguente formula

$$\hat{f} = \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

laddove

$$x_i = a + (i - 1)\delta_x$$

e $a + (n - 1)\delta_x \leq b$. Si implementi:

- una classe astratta `Function` con un metodo `calcola_media` che calcoli la media come richiesto. La funzione prende in input l'intervallo $[a, b]$ ed il numero di sotto-intervalli n da creare;
- una sotto-classe retta, per cui si sa calcolare analiticamente $\hat{f}$;
- una sotto-classe parabola, per cui si sa calcolare analiticamente $\hat{f}$;
- una sotto-classe iperbole, per cui si sa calcolare analiticamente $\hat{f}$;
- un test per utilizzare vari oggetti di tutte le classe definite, controllando che i valori calcolati da `calcola_media` coincidano con quelli analitici.

Esercizio 17.4 (Classe astratta con decoratore)**.** Usando un decoratore e la super-classe `ABC` (abstract base class),

- implementare una classe astratta `Figura`, con i metodi astratti `perimetro` ed `area`;
- implementare una classe `Triangolo` e una classe `Circonferenza` che ereditano da `Figura`;
- implementare il metodo `__init__` che instanzi, nel caso del triangolo, il lato e l'altezza, mentre nel caso della circonferenza il raggio;
- implementare i metodi `perimetro` ed `area` per il triangolo e per la circonferenza.

18 List comprehension

Esercizio 18.1. Si crei una lista di elementi nell'intervallo da 1000 a 2500 con step di 130, utilizzando la list comprehension. Si ragioni se la comprehension serva davvero in questo caso, o meno.

Esercizio 18.2. Si costruisca una funzione che prende in input una lista ed una costante c, e restituisca la lista degli stessi elementi aumentati di c, mediante comprehension.

Esercizio 18.3. Si costruisca una funzione che prende in input una lista ed una costante c, e restituisca la lista degli stessi elementi aumentati di c, mediante comprehension.

Esercizio 18.4. Utilizzando la comprehension, si costruisca una lista dai quadrati di una altra lista solamente se il valore del quadrato supera 100.

Esercizio 18.5. Si testi il seguente codice

```
[x for x in "This is my string"]
```

Utilizzando la comprehension e considerando le seguenti variabili

```
nums = [i for i in range(1,1001)]
```

```
string = "Stringa per l'esercizio pratico di list comprehension."
```

si:

- costruisca la lista dei numeri $x \in [1, 1000]$ divisibili per 8;
- costruisca la lista dei numeri $x \in [1, 1000]$ che contengono la cifra `"6"`;
- contino il numero di spazi nella stringa `string`;
- rimuovano le vocali dalla stringa `string`;
- trovino tutte e sole le parole con meno di 5 lettere nella stringa `string` (per "tokenizzare" una frase nelle parole che la compongano si utilizzi `string.split(" ")`);

- trovino tutti i numeri $x \in [1, 1000]$ che sono divisibili per numeri ad una singola cifra escluso 1 (quindi per 2, ..., 9; si utilizzi una comprehension annidata in questo caso).

19 Iteratori

Esercizio 19.1 (Flusso pari)**.** Si scriva una classe per un iteratore che genera il flusso

$$x, x + \delta, x + 2\delta, x + 3\delta, \ldots, y$$

con x, δ ed $y > x$ input della classe.

Esercizio 19.2 (Flusso primi da lista)**.** Si scriva una classe per un iteratore che restituisce solamente i numeri primi all'interno di una lista di interi (i.e., la classe memorizza la lista, e l'iteratore ci naviga sopra di conseguenza).

Esercizio 19.3 (Flusso da complemento)**.** Si scriva una classe per un iteratore che restituisce solamente i numeri presenti all'interno di una lista di interi x, che non sono presenti all'interno di una altra lista di interi y (i.e., la classe memorizza due liste, e l'iteratore ci naviga sopra di conseguenza).

Esercizio 19.4 (Flusso doppio)**.** Si scriva una classe per un iteratore che genera gli elementi della successione

$$a_i = \frac{2^i}{2i - 1}$$

Esercizio 19.5 (Discretizzazione funzione)**.** Si consideri una funzione $f(x)$ discretizzata sull'intervallo $[a, b]$ sui punti

$$x_1, x_2, \ldots, x_n$$

con $x_i = a + (i - 1)\delta_x$. Si costruisca un iteratore che generi i valori

$$f(x_1), f(x_2), \ldots, f(x_n).$$

Esercizio 19.6 (Calendario itinerante)**.** Realizzare una classe `Date` che presenta come attributi una generica data (giorno, mese, anno).

Inoltre, la classe presenta i seguenti metodi:

- `__init__`, costruttore per creare un oggetto della classe;
- `__str__`, funzione per stampare la data;
- `__iter__`, funzione per creare un iteratore, utilizzato per "muoversi" tra i giorni secondo un incremento di un giorno;

- `__next__`, funzione per realizzare la logica dell'iteratore. In particolare, l'iteratore deve essere in grado di riconoscere se l'anno di riferimento 'e un anno bisestile ed in caso modificare i giorni dei mesi opportuni. L'iteratore deve essere in grado di cambiare mese e/o anno dove 'e necessario (es. da 30/10/2020 si passa a 1/11/2020, oppure da 31/11/2020 si passa a 1/1/2021).

Utilizzare una funzione esterna che determina se l'anno di riferimento sia bisestile oppure no; si osservi che un anno viene considerato bisestile nei due casi seguenti:

- l'anno è divisibile per 400;
- l'anno è divisibile per 4, ma non per 100.

Il corpo principale del programma dovrà testare il corretto funzionamento della classe `Date` su tutte le casistiche che potrebbero essere di interesse.

PART

V

Alcuni esercizi risolti e spiegati

Qui trovi una serie di esercizi, circa uno o due per argomento, con le rispettive soluzioni spiegate in dettaglio per aiutarti nella comprensione dei concetti spiegati nel corso. Prima di leggere le soluzioni, prova a risolverli autonomamente.

Nota bene: in questa sezione utilizziamo sia nel testo che negli pseudocodici presenti il simbolo $\leftarrow$ per indicare l'assegnazione, operatore binario `=` in `C`, e il simbolo $=$ per indicare l'uguaglianza, operatore binario `==` in `C`.

20 Serie e successioni

Esercizio - *Convergenza di una successione.* Implementare una funzione che, dato un numero $n > 0$, calcoli il termine n-esimo della seguente successione:

$$a_i = \begin{cases} \frac{1}{2} & \text{se } i = 1 \\ \frac{a_{i-1} + 1}{2} & \text{se } i > 1 \end{cases}$$

e verificare, usando la funzione scritta, che il limite della successione sia 1.

Esempio di output.

```
> Inserire un numero intero positivo: 3
> Termine numero 3 della successione: 0.8750000
```

Soluzione - *Convergenza di una successione.* L'esercizio ci chiede di calcolare una data successione fino al termine n-esimo dove $n > 0$ viene dato. La formula ci dice che $a_1 = 1/2$ mentre ogni altro termine delle successione con $i > 1$ è calcolato a partire dal termine precedente, cioè usando a_{i-1}. Questo significa che, ad esempio, considerando $i = 2$ si ha:

$$a_2 = \frac{a_1 + 1}{2}$$

e, siccome la formula ci fornisce esplicitamente il valore di a_1, possiamo andare a sostituire per esplicitarne il valore:

$$a_2 = \left(\frac{1}{2} + 1\right)\frac{1}{2} = \frac{3}{4}$$

abbiamo così ottenuto il valore della serie per $i = 2$. Avendo a_2, possiamo ora andare a calcolare il valore della serie per $i = 3$:

$$a_3 = \frac{a_2 + 1}{2} = \left(\frac{3}{4} + 1\right)\frac{1}{2} = \frac{7}{8}$$

e così via. Una cosa che possiamo immediatamente notare è che:

$$a_1 = \frac{1}{2} = 0.5 < a_2 = \frac{3}{4} = 0.75 < a_3 = \frac{7}{8} = 0.875 < 1$$

cioè sembra che che la serie si avvicini incrementalmente a 1, senza però diventarne maggiore. Intutivamente sembra quindi plausibile che la serie converga a 1. Per verificarlo inseririremo un numero $n \gg 1$ nella funzione che andremo a scrivere e verificheremo che il termine a_n che essa restituirà sia circa uguale ad 1.

Facciamo notare che, in generale, per essere veramente certi sul limite di una successione è necessario provarlo usando l'opportuno formalismo matematico. Qui non riporteremo la dimostrazione perchè non è richiesto dall'esercizio.

Vediamo quindi come scrivere un programma che calcoli il termine n-esimo di questa serie. In pratica, ripetiamo i passaggi che abbiamo fatto sopra in un programma. Partiamo quindi definendo una variabile `a_i` che conterrà il termine i-esimo della successione. Poi procediamo così:

1. inizializziamo la variabile `a_i` $\leftarrow 1/2$, che è il valore che del termine i-esimo della successione quando $i = 1$;

2. scriviamo un ciclo `for` per calcolare ogni termine della successione (escluso il primo che abbiamo già calcolato), fino al termine n-esimo. Dobbiamo calcolare ogni termine perchè:

   - per calcolare a_n abbiamo bisogno di a_{n-1};
   - per calcolare a_{n-1} abbiamo bisogno di a_{n-2};
   - ...
   - per calcolare a_4 abbiamo bisogno di a_3;
   - per calcolare a_3 abbiamo bisogno di a_2;
   - per calcolare a_2 usiamo a_1 che conosciamo.

   Vogliamo quindi un ciclo che vada da $i = 1$ a $i = n$ escluso. Scegliamo di usare un ciclo `for` e non un ciclo `while` perchè sappiamo a priori quante iterazioni bisogna fare, cioè $n - 1$ iterazioni;

3. calcoliamo il termine i-esimo della successione e aggiorniamo la variabile `a_i`:

$$\texttt{a\_i} \leftarrow \frac{\texttt{a\_i} + 1}{2}$$

ad ogni iterazione del ciclo `for`. Alla fine delle $n - 1$ iterazioni quindi a_i conterrà il termine n-esimo della successione.

Per capire meglio cosa sta succedendo nell'ultimo punto, proviamo a scrivere esplicitamente ciò che abbiamo calcolato in precedenza come se dovessimo implementarlo in una programma senza usare un ciclo `for`. Lo facciamo prima usando una variabile diversa per ogni termine della successione. Quindi definiamo `a_1`, `a_2` e `a_3` e abbiamo quindi i passaggi:

$$\begin{aligned}
\texttt{a_1} &\leftarrow \frac{1}{2} \\
\texttt{a_2} &\leftarrow \frac{\texttt{a_1} + 1}{2} = \frac{3}{4} \\
\texttt{a_3} &\leftarrow \frac{\texttt{a_2} + 1}{2} = \frac{7}{8}
\end{aligned}$$

e ora facciamo la stessa cosa utilizzando una sola variabile `a_i`:

$$\begin{aligned}
\texttt{a_i} &\leftarrow \frac{1}{2} \\
\texttt{a_i} &\leftarrow \frac{\texttt{a_i} + 1}{2} = \frac{3}{4} \\
\texttt{a_i} &\leftarrow \frac{\texttt{a_i} + 1}{2} = \frac{7}{8}
\end{aligned}$$

come puoi vedere le due procedure sono equivalenti.

Per finire quindi riportiamo lo pseudo-codice di seguito, lasciando al lettore l'implementazione usando un linguaggio di programmazione.

Algoritmo 1: funzione per calcolare il termine n-esimo della successione

```
funzione successione(n):
    a_i ← 1/2

    per i ← 1 a n fai
        a_i ← (a_i + 1)/2
    fine

    ritorna a_i
fine
```

A questo non resta altro che dare in input un numero n sufficentemente grande alla funzione ($n = 100$ dovrebbe essere sufficente) e verificare che ciò che viene restituito è circa uguale a 1: provare per credere!

Domanda: saresti in grado di implementare una funzione ricorsiva che risolve lo stesso problema?

21 Funzioni ricorsive

Esercizio - *Numero primo.* Dato un numero $n \in \mathbb{N}$ tale che $n > 1$ **stabilire se sia primo** implementando una **funzione ricorsiva**.

Esempio di output.

```
> Inserire un numero intero maggiore di 1: 15
> 15 non è primo
```

Soluzione - *Numero primo.* Il primo passo è ricordare la definizione di numero primo. Un numero primo è un numero intero maggiore di 1 che ammette solo divisori banali, cioè 1 e sé stesso. Per completezza, ricordiamo anche una possibile definizione di divisore: m è un divisore di n se il resto dato dalla divisione intera n/m è nullo.

Detto questo, prima di dare una possibile soluzione, notiamo una cosa banale ma fondamentale: un numero intero n non può essere divisibile per un numero intero $m > n$.

Domanda: dato un generico $n > 1$ esiste almeno un altro intervallo di numeri, oltre ad $n > m$, che possiamo escludere? Cioè, esiste almeno un altro intervallo di numeri che sicuramente non divide n? Se sì, quale?

La definizione e l'osservazione ci suggeriscono una soluzione: dato un numero intero $n > 1$ testiamo se qualche numero m appartenente all'insieme $A_n = \{k \in \mathbb{N} : 1 < k < n\} = \{2, 3, ..., n-2, n-1\}$ è un divisore di n. Questo controllo può risultare in 2 eventi esclusivi:

- esiste almeno un $m \in A_n$ tale per cui il resto della divisione n/m è nullo, allora n non è primo;
- altrimenti, per ogni $m \in A_n$ il resto della divisione n/m è diverso da 0, allora n è un numero primo.

Il primo problema che ci poniamo è quindi, dati $n > 1$ ed $m \in A_n$, come controllare se il resto della divisione n/m è 0. Per risolverlo, utilizziamo l'operatore modulo `%` in `C` il quale restituisce il resto della divisione intera tra i due operandi. Questo è illustrato in 45.

Domanda: come implementeresti l'operatore modulo tra due interi utilizzando solamente gli operatori divisione `/` e moltiplicazione `*`?

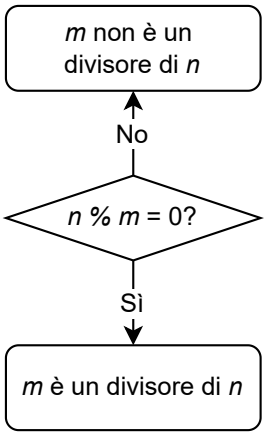


Figura 45. Diagramma di flusso per stabilire se un numero intero $m \in A_n$ è un divisore di un numero intero $n > 1$.

Il secondo problema è come vagliare tutti gli elementi in A_n. Un modo intuitivo è partire da $m \leftarrow n - 1$ e poi decrementare di 1 il valore di m fino ad arrivare a $m = 1$. È analogo partire da $m \leftarrow 2$ e incrementare fino ad $m = n$. Noi qui presentiamo una soluzione partendo da $m \leftarrow n - 1$. Qualsiasi dei due modi si scelga, ad ogni iterazione (esclusa l'ultima) si controlla se n è divisibile per m. Questo può risultare in una delle seguenti 2 possibilità:

- n è divisibile per m, allora esiste almeno un numero diverso da 1 e da n per cui n è divisibile, quindi n non è primo e l'algoritmo termina;
- n non è divisibile per m, allora decrementiamo $m \leftarrow m - 1$ e abbiamo un'altra biforcazione:
  - se rimangono ancora dei numeri in A_n che non abbiamo controllato, cioè $m > 1$, allora ripetiamo lo stesso controllo;
  - altrimenti se abbiamo già considerato e controllato tutti i restanti valori in A_n, cioè $m = 1$, allora n è primo.

La procedura è mostrata in modo più compatto e intuitivo in figura 46. Inoltre, in figura 47 è presente un esempio di funzionamento dell'algoritmo considerando il numero primo $n = 5$.

Ora dobbiamo tradurre tutto questo in codice tenendo in considerazione che l'esercizio chiede di farlo utilizzando una funzione ricorsiva. Quando andiamo a scrivere una funzione ricorsiva dobbiamo occuparci di stabilire:

- il parametro (o i parametri) della funzione;
- la condizione di base (o le condizioni di base) per chiudere la ricorsione;
- un'istruzione (o diverse istruzioni) da eseguire ad ogni iterazione ricorsiva;

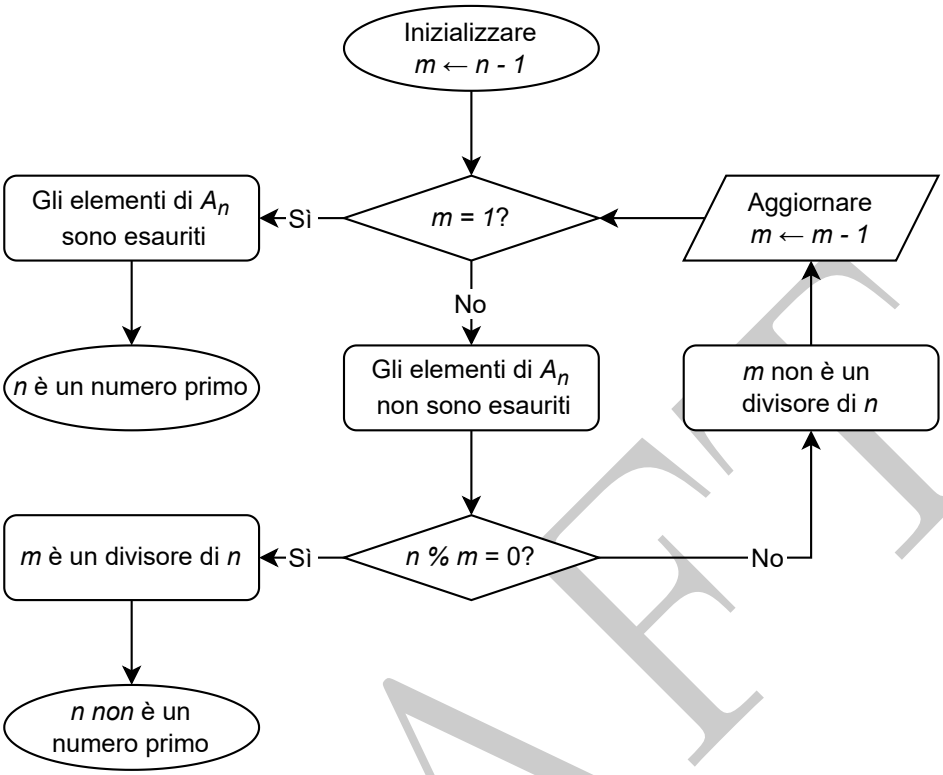


Figura 46. Diagramma di flusso per stabilire se un numero intero n è primo.

- la chiamata ricorsiva (o le chiamate ricorsive) alla funzione con i parametri opportunamente modificati in modo tale che la condizione di base sia prima o poi soddisfatta;
- il valore (o i valori) restituiti della funzione.

Domanda: arrivati a questo punto, saresti in grado di scrivere da solo la soluzione ricorsiva stabilendo ognuno dei 5 punti sopra riportarti? Prima di proseguire leggendo la soluzione, prova a fare almeno un tentativo.

Innanzitutto cominciamo stabilendo ciò che vogliamo che la funzione restituisca: siccome non ci sono specifiche, scegliamo di restituire 0 se il numero n non è primo e 1 se invece è primo.

Dopodichè passiamo a stabilire i parametri: sicuramente abbiamo bisogno di almeno due valori, cioè $n > 1$ e $m \in A_n$. Infatti ad ogni iterazione vogliamo verificare se n è divisibile per m controllando il modulo tra questi due numeri: se il modulo è nullo allora esiste un $m \in A_n$ divisore di n. Perciò non è primo e quindi chiudiamo la ricorsione restituendo 0.

Infine, partendo da $m \leftarrow n-1$, la chiamata ricorsiva sarà fatta decrementando il valore del parametro m di 1 e quindi la condizione di base sarà $m = 1$: se questa condizione è soddisfatta allora significa che abbiamo già chiamato la funzione ricorsivamente per ogni valore in A_n e nessun valore di A_n è risultato un divisore di n. Quindi n è primo e restituiamo 1.

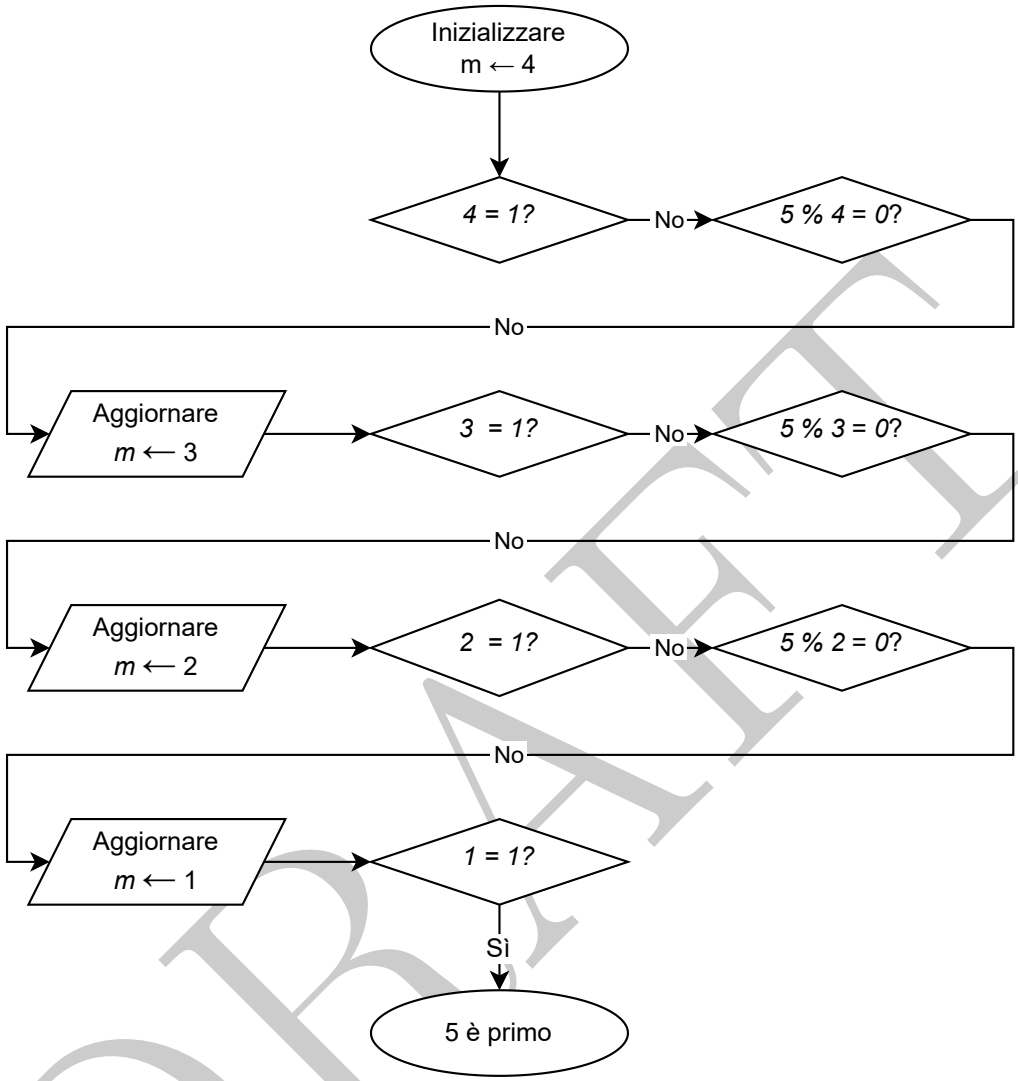


Figura 47. Esempio di funzionamento dell'algoritmo considerando il numero $n = 5$.

Riportiamo ora lo pseudocodice per concludere, lasciando al lettore la traduzione in linguaggio `C`.

Algoritmo 2: funzione ricorsiva per stabilire se un numero intero è primo

```
funzione primo(n, m):
    se m = 1 allora
        ritorna 1                                   /* n è primo */
    fine

    se n % m = 0 allora
        ritorna 0                                /* n non è primo */
    fine

    ritorna primo(n, m − 1)
fine
```

22 Vettori e algoritmi

Esercizio - *Counting sort.* Si scriva una funzione che dato un array `A` di dimensione $n > 0$ che contenga solamente numeri interi non negativi, implementi l'algoritmo di ordinamento *counting sort*. Indicando con $\texttt{A}[i] \in \mathbb{N}$ l'elemento i-esimo dell'array `A` dove $i = 0, \ldots n-1$, l'algoritmo è descritto dai seguenti passaggi:

1. allocare un array `B` di dimensione $k+1$ dove $k = \max\{\texttt{A}[i] : i = 0, .., n-1\}$;
2. contare il numero di occorrenze di (cioè il numero di volte in cui appare) ogni elemento $h \in \{0, 1, ..., k\}$ in `A` e assegnarlo a $\texttt{B}[h]$;
3. utilizzare l'informazione presente in `B` per riordinare gli elementi di `A`.

Soluzione - *Counting sort.* Per capire come trovare una soluzione all'esercizio consideriamo un esempio. La strada che vogliamo percorrere è quella di risolvere un punto alla volta implementando per ognuno un'opportuna funzione. Alla fine potremmo usare le funzioni scritte per risolvere l'intero problema. Supponiamo quindi di avere il seguente array `A`:

A	5	7	5	1	4	1	2	2	5

la prima cosa che ci richiede l'esercizio è di allocare un array `B` di dimensione pari al valore dell'elemento massimo contenuto in `A` cioè, in questo caso, $k = 7$. Dobbiamo quindi innanzitutto trovare il massimo in `A`: lo possiamo fare molto semplicemente con una singola iterazione sugli elementi di `A` come fatto nell'algoritmo 3. In questo algoritmo inizializziamo il massimo al primo valore di `A`, dopodichè iteriamo su tutti gli elementi di `A` aggiornando l'attuale valore del massimo ogni volta che troviamo un elemento maggiore di esso.

Algoritmo 3: funzione per trovare l'elemento massimo di un array

```
funzione massimo(A, n):
    m ← A[0]              /* Variabile che conterrà il massimo */

    per i ← 1 a n fai
        se A[i] > m allora
            m ← A[i]          /* A[i] diventa l'attuale massimo */
        fine
    fine

    ritorna m
fine
```

Fatto questo inizializziamo un array `B` con una lunghezza $m = k + 1$ dove k è il valore ritornato dalla funzione `massimo`. Dopodichè contiamo le occorenze dei valori di `A` e le scriviamo in `B`. Per farlo, poniamo gli elementi di `B` a 0:

B	0	0	0	0	0	0	0	0

e poi, siccome gli elementi di `B` sono in numero uguale al numero di possibili valori diversi che troviamo in `A`, iteriamo una volta sull'array `A` e, all'iterazione i-esima, andiamo a incrementare il valore contenuto in `B`[`A`[i]] di 1. In pratica, nel nostro esempio:

B	0	0	0	0	0	0	0	0

$\downarrow$ `A`[0] = 5

B	0	0	0	0	0	1	0	0

$\downarrow$ `A`[1] = 7

B	0	0	0	0	0	1	0	1

$\downarrow$ `A`[2] = 5

B	0	0	0	0	0	2	0	1

$\downarrow$ `A`[3] = 1

⋮

$\downarrow$ `A`[8] = 5

B	0	2	2	0	1	3	0	1

alla fine di questa procedura `B` conterrà i valori desiderati. L'algoritmo che la implementa è il numero 4.

Algoritmo 4: funzione per contare le occorenze

```
funzione conta_occorenze(A, B, n, m):
    per i ← 0 a m fai
        B[A[i]] ← 0          /* Inizializza gli elementi di B a 0 */
    fine

    per i ← 0 a n fai
        B[A[i]] ← B[A[i]] + 1          /* Conta le occorrenze */
    fine
fine
```

Abbiamo così implementato i primi due punti. Ora ci manca l'ultimo per il quale serve un pochina di intuizione.

Domanda: arrivati a questo punto, hai già capito come risolvere il problema posto del terzo punto? Se la risposta è negativa prova a ragionare sugli array ottenuti finora in quest'esempio:

A	5	7	5	1	4	1	2	2	5

B	0	2	2	0	1	3	0	1

in particolare, prova a confrontare i loro elementi con i rispettivi indici avendo in mente come sarebbe l'array `A` ordinato.

Osserviamo che in `B`$[i]$ con $i > 0$ è contenuto il numero di volte in cui appare il valore i dopo il valore $i-1$ nell'array `A` ordinato. In altre parole, se il valore $i-1$ appare in posizione j nell'array `A` ordinato, allora l'ultima posizione in cui apparirà l'elemento i nell'array ordinato è $j+$`B`$[i]$. Ovviamente `B`$[0]$ ci dice quante volte lo 0 appare a partire dalla prima posizione nell'array ordinato. Quello che facciamo è quindi iterare sull'array `B` e, per ogni `B`$[i]$, aggiungiamo in `A`, in posizione sempre crescente partendo da 0, l'elemento i per un numero di volte pari a `B`$[i]$. Questo è fatto nell'algoritmo 5 e mostrato in seguito.

A	5	7	5	1	4	1	2	2	5

$\downarrow B[0] = 0$

A	5	7	5	1	4	1	2	2	5

$\downarrow B[1] = 2$

A	1	1	5	1	4	1	2	2	5

$\downarrow$ `B`$[2] = 2$

A	1	1	2	2	4	1	2	2	5

$\downarrow$ `B`$[3] = 0$

A	1	1	2	2	4	1	2	2	5

$\downarrow$ `B`$[4] = 1$

$\vdots$

$\downarrow$ `B`$[7] = 1$

A	1	1	2	2	4	5	5	5	7

Domanda: notiamo che `B`$[0] = 0$ perchè 0 non appare mai in `A`. Inoltre 0, essendo minore di qualsiasi elemento in `A`, non apparirà nemmeno nell'array `A` ordinato. Alla luce di questo, riusciresti a riscrivere una versione ottimizzata del codice che, in generale, usa una quantità minore di memoria (cioè che usa un array `B` più corto)?

Algoritmo 5: funzione per ordinare l'array `A` in base al contenuto di `B`

```
funzione ordina(A, B, m):
    k ← 0            /* Indice a cui inseriamo il valore in A */

    per i ← 0 a m fai
        finché B[i] > 0 fai
            A[k] ← i
            k ← k + 1
            B[i] ← B[i] − 1
        fine
    fine
fine
```

Per concludere, riportiamo l'algoritmo completo che utilizza le tre funzioni definite finora.

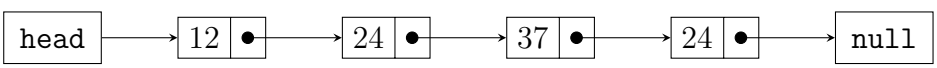
Algoritmo 6: funzione che ordina l'array `A` utilizzando l'algoritmo di counting sort

```
funzione counting_sort(A, n):
    k ← massimo(A, n)
    B ← alloca[k + 1]
    conta_occorenze(A, B, n, k + 1)
    ordina(A, B, k + 1)
fine
```


23 Liste

Esercizio - *Rimuovere duplicati.* Si scriva una funzione che data la testa `head` (cioè il puntatore al primo elemento) di una lista di numeri interi, elimini i nodi con valore duplicato se presenti. Cioè, se sono presenti due o più nodi con lo stesso valore $n \in \mathbb{N}$ elimini uno o più di quei nodi in modo che alla fine ne rimanga solamente uno contenente il valore n.

Soluzione - *Rimuovere duplicati.* Supponiamo di avere un puntatore `head` al primo nodo, contenente il valore 12, della seguente lista ordinata di interi.

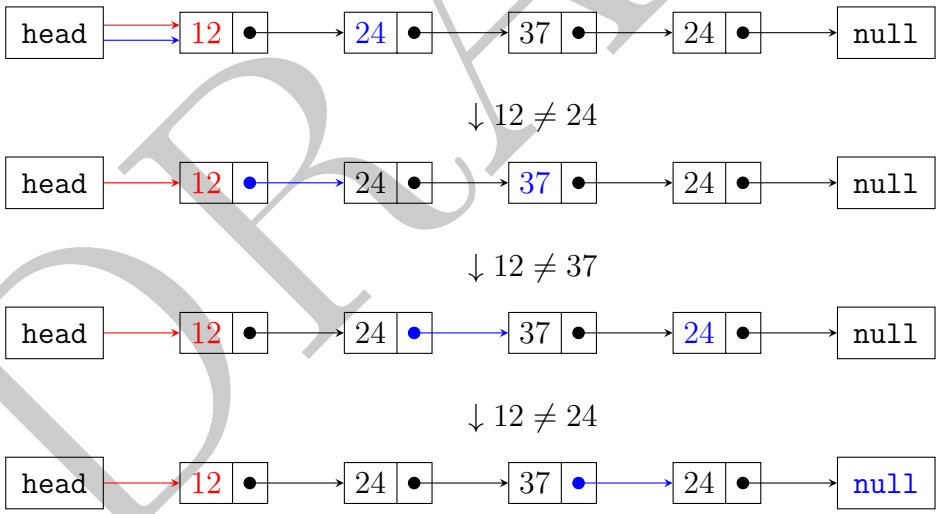


Avendo una visione completa della lista, è facile per noi vedere che il secondo e il quarto nodo contengono uno stesso valore pari a 24. Tuttavia nel nostro programma non abbiamo questa visione globale ma dobbiamo arrangiarci

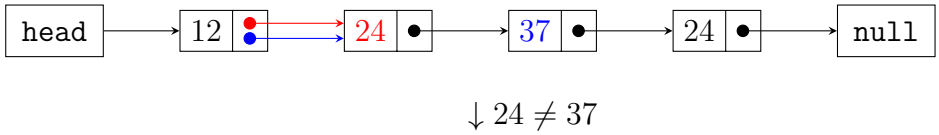
usando solamente dei puntatori a nodi successivi. Abbiamo due problemi da risolvere:

- come inviduare nodi contenente lo stesso valore;
- come eliminare un nodo dalla lista e, dati due nodi con lo stesso valore, quale dei due eliminare.

Cominciamo dal primo punto. Per risolverlo abbiamo bisogno di due puntatori: uno punterà al nodo contenente il valore per il quale stiamo controllando che non ci siano duplicati (che chiameremo `rosso` perchè di colore rosso nello schema sottostante) e il secondo (che chiameremo `blu`) invece itererà sulla lista in cerca di un possibile duplicato. All'inizio del programma questi due puntatori saranno entrambi uguali ad `head`, quindi punteranno entrambi al primo nodo contenente il valore 12 e un puntatore al secondo nodo. Dopodichè `blu` comincerà a scorrere la lista in cerca di un duplicato fino a che non la esaurisce. Ad ogni iterazione, confrontiamo il valore del **nodo successivo** rispetto al nodo puntato da `blu` (lo possiamo fare perchè ogni nodo contiene un puntatore al nodo successivo) con il valore del nodo puntato da `rosso`.



Quando `blu` esaurisce la lista, in questo caso senza trovare duplicati, allora facciamo puntare `rosso` al secondo nodo (in generale lo faremo puntare al nodo successivo). Nota che `blu` esaurisce la lista quando il nodo a cui punta contiene un puntatore `null` al nodo successivo. Dopodichè riportiamo indietro `blu` facendolo puntare anch'esso allo stesso nodo puntato da `rosso`, cioè il secondo nodo in questo caso. Possiamo evitare di far ripartire `blu` da `head` siccome, se supponiamo di aver implementato correttamente anche la rimozione dei duplicati come faremo, sappiamo che nei nodi precedenti a quello attualmente puntato da `rosso` non ci saranno duplicati. A questo punto ripetiamo la stessa operazione precedente.



↓ 24 ≠ 37

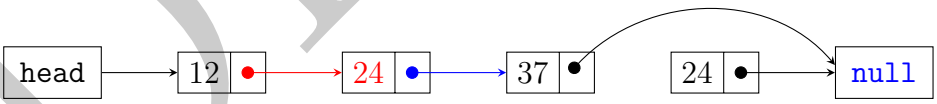
In questo caso abbiamo trovato un duplicato di 24 nel quarto nodo, come ci aspettavamo. Abbiamo quindi capito come fare a trovare i duplicati. Quindi, con le informazioni e i puntatori che abbiamo a disposizione a questo punto dell'esecuzione, pensiamo a come fare a rimovere uno dei due nodi e quale rimuovere.

Domanda: quali dei due nodi rimuoveresti? Perchè?

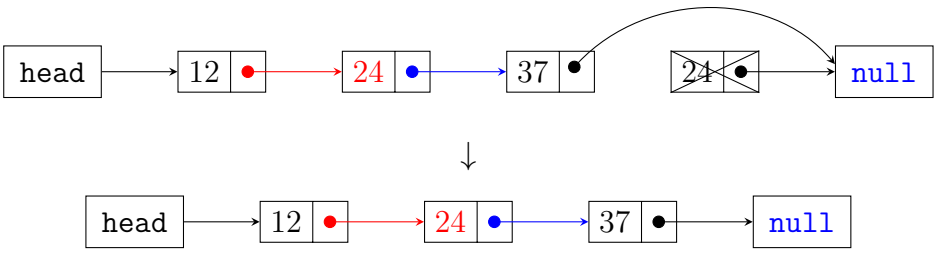
Innanzitutto, è più comodo e intuitivo rimuovere il quarto nodo. Ovvero, in generale, rimuovere il nodo successivo rispetto a quello puntato da `blu`. Questo perchè vogliamo che `rosso` rimanga fermo e ci dica per quale valore della lista stiamo cercando e rimuovendo i duplicati. Se rimuovessimo il nodo puntato da `rosso` allora dovremmo spostare quest'ultimo e farlo puntare al quarto nodo: un'operazione aggiuntiva che non è necessaria.

Domanda: sapresti come fare a rimuovere un nodo dalla lista? Questo è il momento buono per fermarti a pensare a come fare se non lo sai già.

Ora, quello che dobbiamo fare per rimuovere il nodo successivo a quello puntato da `blu` è quello di modificare il puntatore del terzo nodo (cioè del nodo puntato da `blu`) in modo che punti al nodo puntato dal quarto nodo (`null` in questo caso). Facciamo questo perchè, in generale, vogliamo mantenere intatto il collegamento della lista tra la parte precedente e quella successiva di un nodo che vogliamo eliminare.

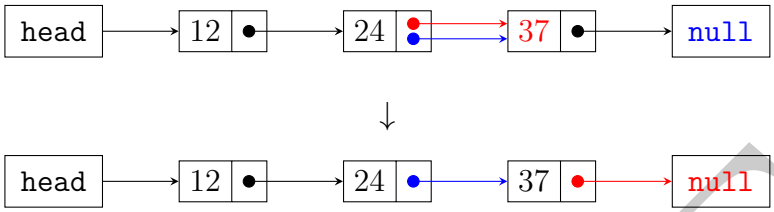


A questo punto possiamo semplicemente eliminare dalla memoria il quarto nodo. Per fare questo dobbiamo salvare una copia del puntatore ad esso prima di spostare il puntatore contenuto del terzo nodo. Andremo poi a liberare la memoria puntata da tale copia per rimuovere il quarto nodo.



A questo punto il nostro algoritmo prosegue: siccome `blu` ha esaurito la lista allora il processo ricomincia e stavolta sia `blu` che `rosso` punteranno al terzo nodo. Ma il nodo successivo al terzo è `null` quindi `blu` non deve fare alcuna iterazione. Modifichiamo di nuovo rosso e scopriamo è `null`. Questo ci dice

che abbiamo cercato e rimosso i duplicati per ogni possibile valore presente nella lista, quindi l'algoritmo termina.



La generalizzazione della soluzione spiegata in quest'esempio è riportata nell'algoritmo 7.

Algoritmo 7: rimuove i duplicati della lista con testa *head*

```
funzione rimuovi_duplicati(head):
    rosso ← head

    finché rosso ≠ null fai
        blu ← rosso                          /* Riporto indietro blu */

        finché (blu → next) ≠ null fai
            se rosso → valore = blu → valore allora
                copia ← (blue → next)          /* Copio il puntatore */
                (blue → next) ← (blue → next → next)     /* Sposto */
                free(copia)                          /* Rimuovo il nodo */
            altrimenti se rosso → valore ≠ blu → valore allora
                blu ← (blu → next)             /* Faccio avanzare blu */
            fine
        fine

        rosso ← (rosso → next)             /* Faccio avanzare rosso */
    fine
fine
```

Esercizio - *Appendere ad un lista alcuni elementi di un array.* Si scriva una funzione che dato un array `A` di dimensione n contenente solamente interi maggiori di 1, crei una lista e vi appenda solamente gli elementi `A`$[i]$ con $i = 0, 1, \ldots, n-1$ tali che:

$$\forall j \in \{0, 1, ..., n-1\} \setminus \{i\},\ \texttt{A}[i] \mod \texttt{A}[j] \neq 0$$

dove mod è l'operazione di modulo e $\setminus$ la differenza tra insiemi. Si fa notare che *appendere* un elemento ad una lista significa aggiungerlo in fondo ad essa.

Per la soluzione del problema si richiede inoltre di rispettare due vincoli (che rendono l'algoritmo efficente in termini di memoria usata e di complessità logaritmica):

- non è possibile allocare e usare altri array oltre ad `A`;
- per aggiungere un elemento alla fine della lista non si può reitare la lista. In altre parole, un elemento deve essere aggiunto senza prima scorrere su tutti gli altri elementi.

La funzione deve restituire un puntatore al primo elemento (nodo) della lista.

Soluzione - *Appendere ad un lista alcuni elementi di un array.* Cominciamo da alcune osservazione banali ma fondamentali. Dire che dato un certo i dobbiamo soddisfare

$$\forall j \in \{0, 1, ..., n-1\} \setminus \{i\}$$

una data condizione equivale a dire che dobbiamo soddisfare tale condizione **per ogni** $j = 0, \ldots, n-1$ tale che $j \neq i$. Quindi, in questo caso, dobbiamo cercare tra i valori $i \in \{0, \ldots, n-1\}$ quelli tali che per ogni $j \neq i$ sia soddisfatta:

$$\texttt{A}[i] \mod \texttt{A}[j] \neq 0$$

ovvero che il resto della divisione $\texttt{A}[i]/\texttt{A}[j]$ non sia nullo, che equivale a dire che $\texttt{A}[i]$ non è divisibile per $\texttt{A}[j]$. Perciò, a parole, l'esercizio richiede di appendere alla lista solamente gli elementi di `A` che non sono divisibili per ogni altro elemento di `A`. Notiamo infine che l'operazione di modulo è implementata dall'operatore binario `%` in `C`.

Con questo in mente, abbiamo due problemi, che posso essere considerati indipendenti, da risolvere:

1. trovare gli elementi di `A` che soddisfano la condizione;
2. creare e appendere un elemento ad una lista in modo efficente, cioè soddisfando le due richieste.

Andiamo per ordine risolvendo il primo dei due punti. Per trovare gli elementi che soddisfano la condizione cambiamo punto di vista cercando quelli che non la soddisfano. Per capire quali elementi non soddisfano la condizione dobbiamo negarla. Otteniamo così che un elemento $\texttt{A}[i]$ con $i = 0, \ldots, n-1$ non soddisfa la condizione richiesta se

$$\exists j \in \{0, 1, ..., n-1\} \setminus \{i\} : \texttt{A}[i] \mod \texttt{A}[j] = 0$$

quindi, dato i, iteriamo su tutti gli elementi di `A` escluso `A`$[i]$, e:

- se troviamo un elemento `A`$[j]$ con $j \neq i$ tale che `A`$[i]$ mod `A`$[j] = 0$ allora l'elemento non soddisfa la condizione (quindi non lo appenderemo alla lista);
- se esauriamo tutti gli elementi e non ne troviamo almeno uno tale che `A`$[i]$ mod `A`$[j] = 0$ allora `A`$[i]$ soddisfa la condizione (e quindi lo appendiamo alla lista).

Iniziamo quindi scrivendo una funzione che dati `A` e l'indice i ritorni 0 se `A`$[i]$ non soddisfa la condizione e 1 se la soddisfa.

Algoritmo 8: funzione per controllare se un elemento di `A` soddisfa la condizione

```
funzione controlla_condizione(A, i, n):
    per j ← 0 a n fai
        se A[i] % A[j] = 0 allora
            se j ≠ i allora
                ritorna 0          /* Condizione non soddisfatta */
            fine
        fine
    fine

    ritorna 1                      /* Condizione soddisfatta */
fine
```

Concentriamoci ora su come creare una lista e appendervi un elemento in modo efficente. Innanzitutto, supponiamo di aver definito un'opportuna struttura dati `nodo` che contiene un valore intero chiamato `value` e un puntatore `next` ad un `nodo`. In secondo luogo, discutiamo brevemente le due condizioni sull'implementazione richieste dall'esercizio:

- il fatto che non possiamo usare nessuno altro array ci costringe ad appendere gli elementi che soddisfano la condizione man mano che li troviamo nell'array `A`. Quindi, ad esempio, non possiamo prima copiare tutti gli elementi che soddisfano la condizione in un altro array e poi aggiungerli in una sola volta alla lista (cosa che banalmente soddisferebbe la seconda condizione);
- la richiesta di non reiterare ogni volta sulla lista, senza quindi partire ogni volta dalla sua testa, ci obbliga a salvare e usare l'informazione su qual'è l'ultimo elemento della lista. A quest'ultimo andremmo ad appendere il prossimo elemento che troviamo soddisfare la condizione. Dopodichè l'elemento appeso diventerà l'ultimo e quindi dovremmo aggiornare opportunamente le informazioni.

Fatte queste premesse, per facilitarci le cose in seguito definiamo ora una funzione che, dato un valore x crei un nodo con tale valore e restituisca un puntatore ad esso. Vogliamo cioè che la funzione:

1. allochi un nuovo nodo in memoria;
2. inizializzi i suoi membri `value` e `next` rispettivamente a x e `null`;
3. restituisca un puntatore a questo nodo.

Algoritmo 9: alloca un `nodo` contenente un valore x e restituisce un puntatore ad esso

---

funzione `nuovo_nodo`(x)**:**
    $tmp \leftarrow$ **alloca**($nodo$) `/*` *tmp* `è un puntatore al nuovo` *nodo* `*/`
    $(tmp \rightarrow value) \leftarrow x$
    $(tmp \rightarrow next) \leftarrow null$
    ritorna *tmp*
fine

L'idea è quindi iterare una volta sull'array `A` controllando per ogni elemento se la condizione è soddisfatta:

- se la condizione è soddisfatta allochiamo un nuovo nodo usando la funzione definita e modifichiamo il puntatore dell'ultimo nodo della lista in modo che punti al nodo creato;
- se la condizione non è soddisfatta consideriamo l'elemento successivo.

L'iterazione può essere banalmente implementata con un ciclo `for`. Tuttavia ci sono due questioni da considerare:

1. come sappiamo ad ogni iterazione qual'è l'ultimo nodo e come aggiorniamo quest'informazione ad ogni iterazione?
2. come e quando creiamo il primo nodo che sarà la testa della lista non sapendo, a priori, quale sarà il primo elemento dell'array `A` che soddisfa la condizione?

Domanda: fermati a ragionare su queste domande, riesci a vedere dov'è il problema e come risolverlo?

È ovvio che dovremmo avere due diversi puntatori `head` and `tail` i quali puntano rispettivamente al primo ed ultimo nodo della lista. Iniziliammo `head` a `null` (come vedremo non è necessario farlo per `tail`) prima di cominiciare a iterare `A`. Inoltre, ragionando sulle domande che ci siamo appena posti intuiamo che dobbiamo comportarci in modo diverso quando incontriamo per la prima volta un elemento di `A` che soddisfa la condizione o se ne abbiamo già incontrato uno in precedenza. In particolare abbiamo che:

- se è la prima volta allora allochiamo un nuovo nodo con il valore dell'array che soddisfa la condizione e lo facciamo puntare sia da `head` che da `tail`:
- altrimenti se non è la prima volta:
  1. creiamo un nuovo nodo con il valore dell'array che soddisfa la condizione;
  2. appendiamo il nuovo nodo alla coda della lista (cioè lo facciamo puntare dal puntatore `next` dell'elemento puntato da `tail`);
  3. aggiorniamo `tail` facendolo puntare al nodo appena creato.

Notiamo che chiedersi se incontriamo o meno per la prima volta un elemento o meno equivale a chiedersi se abbiamo già inizializzato o meno la testa della lista. Cioè se la `head` è uguale a `null` oppure no.

Riportiamo quindi l'algoritmo 10 che implementa la soluzione usando le funzione sopra definite.

Algoritmo 10: dato una array `A` crea una lista contenente solamente gli elementi di `A` che non sono tra loro divisibili

```
funzione crea_lista(A, n):
    head = null

    per i ← 0 a n fai
        controllo = controlla_condizione(A, i, n)

        se controllo = 1 allora
            se testa = null allora
                head ← nuovo(A[i])
                tail ← head
            altrimenti se testa ≠ null allora
                tail ← nuovo(A[i])
                tail ← (tail → next)
            fine
        fine
    fine

    ritorna head
fine
```