Pseudocodice(4.0)

Una guida completa alla **lettura e comprensione** degli esercizi di programmazione delle selezioni scolastiche

Lo Staff delle OII

Ottobre 2022

Sullo pseudocodice nella selezione scolastica

Da anni, gli esercizi della selezione scolastica delle OII sono suddivisi in tre parti:

- esercizi di carattere logico-matematico;
- esercizi di programmazione;
- esercizi di carattere algoritmico.

A partire dalla selezione scolastica di novembre 2018 abbiamo introdotto un importante cambiamento relativo alla forma in cui gli **esercizi di programmazione** vengono proposti: invece di fornire del codice vero e proprio (in due dei linguaggi più "popolari" tra quelli insegnati in Italia a livello scolastico: C e Pascal) abbiamo cominciato a utilizzare uno **pseudocodice**. Con *pseudocodice* intendiamo un linguaggio che permette di descrivere programmi usando una sintassi naturale, "umana", senza le rigide regole di un linguaggio di programmazione.

Lo stile di pseudocodice che abbiamo adottato è un'evoluzione di quello già in uso alle *Olimpiadi del Problem Solving*, migliorandolo con vari aiuti visuali e alcune sintassi più intuitive. Ricordiamo comunque che la sintassi dello pseudocodice non è da considerare esaustiva: è possibile che alcuni esercizi utilizzino dei costrutti non documentati in questa guida. Se questo accadrà, i "nuovi" costrutti saranno sempre spiegati nel testo dell'esercizio e pensati per essere facilmente comprensibili.

Attenzione! Questo cambiamento si riferisce **solo agli esercizi di programmazione** contenuti nella **selezione scolastica**. La fase territoriale e nazionale rimangono invariate e richiedono sempre la scrittura di programmi "veri".

È importante notare che, sebbene le fasi successive alla scolastica facciano uso di linguaggi veri e propri, in questo caso esiste una fondamentale distinzione: ciò che valutiamo in questa gara infatti è l'abilità nel **leggere e capire** del codice già scritto, non quella di **scriverne**. Siamo convinti che lo pseudocodice sia particolarmente adatto allo scopo di questa gara.

Con lo pseudocodice, infatti, gli esercizi di programmazione diventano alla portata di tutti gli studenti, non necessariamente di quelli che sanno già programmare, o il cui curriculum scolastico comprende l'informatica: è sufficiente avere curiosità e voglia di imparare e mettersi in gioco!

Novità dell'edizione 2022

Quest'anno abbiamo rinnovato la guida, espandendola per una documentazione più comprensiva dello pseudocodice, con spiegazioni dettagliate e allo stesso tempo comprensibili. Inoltre, lo pseudocodice stesso ha subito delle modifiche:

- sono stati introdotti i cicli `for`;
- non esistono più le procedure, ma solo funzioni, che possono eventualmente non restituire nulla;
- il comando **output** è stato sostituito dalla funzione `output`;
- è stata introdotta la possibilità di "scambiare" il contenuto di due variabili;
- la formattazione dello pseudocodice è stata migliorata, usando anche dei colori per differenziare le varie componenti (parole chiave, variabili, etc.).

Guida rapida allo pseudocodice

Nella pagina successiva trovi un *cheat sheet*, una guida di riferimento sullo pseudocodice che riassume in una tabella i punti salienti della sintassi. Ogni riga della tabella contiene la sintassi di un elemento dello pseudocodice, una breve descrizione, ed eventualmente uno o più esempi del suo utilizzo. Questa guida rapida sarà anche fornita per consultazione insieme al testo della selezione scolastica.

Pseudocodice	Descrizione	Esempio
■ Variabili e tipi		
variable i: *integer*	Dichiarazione della variabile di tipo intero chiamata i	
variable arr: *integer[]*	Dichiarazione di una variabile di tipo array di interi chiamata arr	
{var} ← {espr}	Assegnamento del valore dell'espressione {espr} alla variabile {var}	i ← 1 a ← 3 × i + 5 arr ← [3, 5/a, 2]
arr[{espr}]	Variabile corrispondente all'elemento dell'array arr di indice {espr}	a ← arr[3] + 1 arr[x + 1] ← 2
(a, b) ← (b, a)	Scambio del valore delle variabili a e b	
■ Operatori		
+, −, ×, /, **mod**	Aritmetica: addizione, sottrazione (o negazione), moltiplicazione, divisione intera, resto della divisione intera (modulo)	a + b −a i ← a **mod** 10
==, ≠, <, ≤, >, ≥	Confronto: uguale, diverso, minore, minore o uguale, maggiore, maggiore o uguale	a == 7 2 × (x + 1) ≤ y
and, **or**, **not**	Operatori logici: e, o, non	a > b **and** b ≠ -1 **not** (a > 2 **or** a == 0)
■ Strutture di controllo		

if {condizione} **then**
  {corpo if}
else
  {corpo else}
end if
	Struttura condizionale `if ... else`: se {condizione} è vera viene eseguito {corpo if}, altrimenti viene eseguito {corpo else}.
	La parte **else** può essere omessa
		if n **mod** 2 == 0 **then**
		  i ← 0
		else
		  i ← n − 1
		end if

while {condizione} **do**
  {corpo}
end while
	Ciclo `while`: il blocco {corpo} viene ripetuto fintanto che {condizione} è vera	**while** i < n **do**
		  sum ← sum + i
		  i ← i + 7
		end while

for {indice} **in** {intervallo} **do**
  {corpo}
end for
	Ciclo `for`: il blocco {corpo} viene eseguito mentre la variabile {indice} itera sui valori in {intervallo}, specificato come $[a \dots b)$, che significa "tutti i numeri da a (incluso) fino a b (escluso)"	**for** i **in** [0 ... n) **do**
		  arr[i] ← -1
		end for
		(assegna -1 a tutti gli elementi di un array arr di lunghezza n)

■ Funzioni		

function
fun(var1: *tipo1*, var2: *tipo2*, ...)
→ *ritorno*
  {corpo}
end function
	Funzione con parametri var1, var2, etc.
	Il tipo di ritorno → *ritorno* può essere omesso.
	Il valore viene restituito tramite la parola chiave **return**
		function
		add(a: *integer*, b: *integer*) →
		integer
		  return a + b
		end function

fun() fun(arg1, arg2, ...)	Chiamata alla funzione fun (rispettivamente senza argomenti e con argomenti). La funzione output stampa il valore di una variabile oppure una stringa fissata. Le funzioni min e max restituiscono risp. il minimo e il massimo di due interi	**return** add(a, b) m ← very_big_integer() output(x) output("string") min(a, b) max(a, b)

Documentazione dello pseudocodice

In questa sezione della guida definiamo la **sintassi** dello pseudocodice. Vale a dire, forniamo le nozioni necessarie per la lettura e comprensione di un programma scritto in pseudocodice. Le spiegazioni saranno abbastanza "formali" da essere precise, ma non tecniche, e saranno tutte corredate da piccoli esempi. Nella sezione successiva, invece, troverai molti altri esempi più completi, alcuni dei quali tratti da selezioni scolastiche passate. Dopo aver letto la documentazione, dovresti essere in grado di comprendere il funzionamento di questi programmi:

Programma 1 FizzBuzz

```
1:  function fizzbuzz(n: integer)
2:      variable i: integer
3:      i ← 1
4:      while i ≤ n do
5:          if i mod 3 == 0 then
6:              if i mod 5 == 0 then
7:                  output("fizzbuzz")
8:              else
9:                  output("fizz")
10:             end if
11:         else
12:             if i mod 5 == 0 then
13:                 output("buzz")
14:             else
15:                 output(i)
16:             end if
17:         end if
18:         i ← i + 1
19:     end while
20: end function
```

Programma 2 Uso degli array

```
1:  function maximum(n: integer, v: integer[])
    → integer
2:      variable m: integer
3:      m ← v[0]
4:      for i in [0 ... n) do
5:          if v[i] > m then
6:              m ← v[i]
7:          end if
8:      end for
9:      return m
10: end function
```

Componenti di un programma

Osservando i programmi 1 e 2 qui sopra, puoi notare che compaiono termini e simboli evidenziati in modo diverso: colori diversi, alcuni in grassetto e altri no, etc. Questi "blocchi fondamentali", che esamineremo uno per uno, sono i seguenti:

- **parole chiave** (esempio: **while**, **end**, **variable**)
- variabili (esempio: m)
- *tipi* (esempio: *integer*, *integer[]*)
- valori (esempio: 1, 42)
- **operatori** (esempio: >, **mod**)

È anche evidente che alcune righe di codice sono spostate a destra (si dice che sono *indentate*). Una serie di righe consecutive con lo stesso *livello di indentazione* (cioè, precedute dallo stesso spazio bianco) si chiama **blocco**. I blocchi rappresentano parti di codice che vengono eseguite "consecutivamente" all'interno di una **funzione** o di una **struttura di controllo**. Ciascun blocco è prededuto da una riga che inizia con una parola chiave, ed è seguito da una riga che inizia con la parola chiave **end**.

Parole chiave

Le **parole chiave** sono parole che hanno un significato specifico all'interno di un programma, e, in quanto tali, non possono essere usate altrove (ad esempio, come nome di variabili — vedi sezione successiva). Sono formattate in **blu e grassetto**. Questa è la lista delle parole chiave:

- **function**
- **return**
- **if**
- **then**
- **else**
- **for**
- **in**
- **while**
- **do**
- **end**
- **variable**

La funzione di ciascuna parola chiave verrà esaminata quando parleremo del contesto in cui compare.

Variabili e tipi

Le **variabili** sono fondamentali in un programma. Nel nostro pseudocodice sono formattate in **azzurro e grassetto**.

Una variabile è una sorta di contenitore per un valore, che può essere un numero, o qualcosa di più complicato come una "sequenza di numeri". Queste "tipologie" di variabile si chiamano, appunto, **tipi**, e sono formattati in *blu e corsivo*. Esiste un solo tipo semplice contemplato dal nostro pseudocodice:

- *integer*: è il tipo che rappresenta numeri interi, ovvero $\{\dots, -2, -1, 0, 1, 2, \dots\}$.

Esiste una categoria di tipi "composti", gli **array**, che sono trattati alla fine di questa sezione. **Attenzione!** Alcuni degli esempi che seguono usano gli array, quindi può essere una buona idea riprenderli dopo aver letto il paragrafo su di essi.

Una variabile è identificata da un nome: a variabili diverse corrispondono nomi diversi. Un programma usa le variabili per manipolare valori che possono cambiare durante l'esecuzione del codice. Le variabili possono essere:

- *dichiarate*. Dichiarare una variabile è obbligatorio, e vuol dire comunicare al programma che tale variabile esiste. Per questo, va fatto prima di qualunque altra operazione che coinvolge la variabile. Una dichiarazione di variabile inizia con la parola chiave **variable**, seguita dal nome della variabile, dai due punti, e dal tipo della variabile.
  Attenzione! Appena dopo la dichiarazione, la variabile non contiene nessun valore: deve essere *inizializzata* assegnandone uno.

  Questi sono esempi di dichiarazione:

  variable i: *integer*
  variable arr: *integer[]*

  Queste **non** sono dichiarazioni:

  variable i
  arr: *integer[]*
  integer a

- *inizializzate e modificate tramite assegnamento*. L'assegnamento è infatti l'operazione che assegna un nuovo valore alla variabile. Un assegnamento è costituito dal nome della variabile, seguito dall'operatore $\leftarrow$ (tratteremo gli operatori più avanti), seguito dal valore da assegnare. Quest'ultimo può essere un valore semplice, un'altra variabile, o più in generale un'**espressione** (anche di espressioni parleremo in seguito).

Questi sono esempi di assegnamenti validi:

$$\mathsf{i} \leftarrow 1$$
$$\mathsf{a} \leftarrow 3 \times \mathsf{i} + 5$$
$$\mathsf{arr} \leftarrow [3,\ 5/\mathsf{a},\ 2]$$

Questo **non** è un assegnamento:

$$\mathsf{i} == 1$$

Questo **non** è codice valido:

$$1 \leftarrow 3$$

- *usate nelle espressioni*. Per esempio, una variabile può essere sommata a un'altra variabile, o a un valore, o un'altra espressione, tramite l'operatore $+$. Oppure può essere confrontata, tramite operatori quali $==$, $<$, $\geq$, e altri. Tutto ciò verrà trattato meglio quando parleremo di operatori e di espressioni.

Array

Un array è una sequenza di valori tutti dello stesso tipo, che può avere lunghezza arbitraria (eventualmente lunghezza 0, e in tal caso si parla di *array vuoto*). Il tipo corrispondente si indica con il tipo base, seguito da parentesi quadre: *integer[]* è l'unico tipo array in questo pseudocodice, poiché *integer* è l'unico tipo base.

Il contenuto dell'array si indica con una lista di valori (o espressioni) inframezzati da virgole e racchiusi da parentesi quadre: `[3, i, 1 - 2]`, `[100]`, `[]` (array vuoto).

Attenzione! Un array di lunghezza n è indicizzato, cioè numerato, da 0 a $n-1$. L'indice i corrisponde all'elemento in posizione i. Per esempio, nell'array `[3, -1, 8]`, gli indici sono 0, 1 e 2, e l'indice 1 corrisponde al valore `-1`.

Gli elementi che costituiscono un array sono a loro volta variabili che possono essere usate e assegnate. Per accedere ad un elemento di un array, si scrive il nome dell'array, seguito dall'indice racchiuso da parentesi quadre: `arr[3]`, `arr[i + 1]`.

Nota che l'indice può essere un numero ma anche un'espressione (contenente anche delle variabili). Una volta acceduto ad un elemento, lo si può usare in espressioni e gli si possono assegnare valori: `arr[0] ← 0`, `arr[i] ← arr[i] + 1`.

Approfondimento: il caso delle stringhe. Forse hai notato che, nel programma 1, compaiono delle parole racchiuse da apici doppi (virgolette). Ne

incontrerai ancora in seguito. Si tratta di quelle che in informatica vengono chiamate *stringhe*, cioè sequenze di caratteri. Nella maggior parte dei linguaggi di programmazione esiste un tipo associato alle stringhe, ma nel nostro pseudocodice non esiste un tipo stringa: le stringhe esistono soltanto come possibili argomenti della funzione output, esattamente come nel programma FizzBuzz.

Valori

Un **valore** è un'entità fissa, costante, che compare nel codice in quanto tale: un valore non può essere modificato. Ogni valore deve appartenere a uno dei tipi descritti in precedenza. Esistono quindi valori interi, di tipo *integer*, e valori di tipo array *integer[]*. Nel nostro pseudocodice i valori sono formattati in giallo.

Questi sono esempi di uso dei valori:

```
a ← 0
1 + 7
```

Questa **non** è un'operazione valida:

```
[] ← [3, -1, 8]
```

Attenzione! In questa guida, a volte usiamo la parola *valore* per riferirci al risultato di un'espressione (per esempio, 3×5 risulta nel valore 15), che è un'altra cosa (non è un'entità "scritta" nel codice). Quindi fai attenzione, il significato della parola deve essere talvolta dedotto dal contesto.

Operatori

Un **operatore** è un simbolo, o una parola, che combina i risultati di due espressioni (valori), producendo un nuovo risultato (valore). Gli operatori sono formattati in **viola e grassetto** (o solo viola per i simboli).

Esistono varie classi di operatori:

- *operatori aritmetici*: $+$, $-$, $\times$, $/$, **mod** (addizione, sottrazione, moltiplicazione, divisione intera, resto della divisione intera). Questi prendono due interi

e restituiscono un intero. Il significato dei primi tre dovrebbe essere chiaro. L'operatore / di divisione intera produce come risultato la *parte intera* (cioè, approssimazione verso lo 0) del quoziente di due interi: ad esempio, 7 / 3 restituisce 2 (perché $7/3 = 2.333...$). L'operatore modulo **mod** produce come risultato il resto della divisione di due interi: ad esempio, 7 **mod** 3 restituisce 1. Infine, l'operatore − può essere anche anteposto ad un'espressione, cambiandone il segno (esempio: se il valore di **a** è 3, il valore di −**a** è −3).

- *operatori di confronto*: ==, ≠, <, ≤, >, ≥ (uguale, diverso, minore, minore o uguale, maggiore, maggiore o uguale). Come dice il nome, confrontano due interi. Ad esempio, 1 == 3 + (-2) è vera, mentre 7 < 7 è falsa.
  Attenzione! Il simbolo di uguaglianza è proprio quello che vedi: due uguali "normali" uno dopo l'altro, separati da uno spazio. L'uguale singolo non viene usato in questo pseudocodice, per evitare confusioni con l'assegnamento.

- *operatori logici*: **and**, **or**, **not**. I primi due operatori combinano due espressioni (termini) contenenti operatori di confronto, o altri operatori logici. **and** restituisce vero se entrambi i termini sono veri, e falso altrimenti; **or** restituisce vero se almeno uno dei termini è vero, e falso altrimenti. **not** è un cosiddetto operatore *unario*, perché agisce su una sola espressione. Restituisce vero se l'espressione è falsa, e falso se l'espressione è vera. Per esempio: (**a** == **b**) **and** (**a** ≠ **b**) è falsa qualunque siano i valori di **a** e **b** (perché?), mentre **not** (7 ≥ 9) è vera.

Approfondimento: valori di verità. Nella descrizione degli operatori di confronto e logici, abbiamo parlato di "vero" e "falso". Questi sono detti *valori di verità*, e presuppongono l'esistenza di un tipo opportuno, presente in tutti i linguaggi veri. Il nostro pseudocodice non contempla l'esistenza esplicita di questo tipo (proprio come per le stringhe): i valori di verità possono essere usati solo nei punti decisionali delle strutture di controllo (che vedremo più avanti).

La precedenza tra gli operatori aritmetici è quella tradizionale: × e / hanno precedenza maggiore di + e −. Per quanto riguarda **mod**, questo ha la stessa precedenza di moltiplicazione e divisione, perciò useremo sempre le parentesi tonde per non creare ambiguità. Gli operatori di confronto hanno meno precedenza di quelli aritmetici, ma più precedenza di quelli logici. Anche qui, comunque, useremo parentesi dove necessario per prevenire ambiguità.

Espressioni

Un'**espressione** è un pezzo di codice dotato di un proprio valore, cioè che può essere *valutato*. Ad esempio, sia i valori sia le variabili di cui abbiamo parlato prima sono semplici espressioni. Ma espressioni sono anche cose più complesse, come $(\mathsf{i} + \mathsf{j} \times 3)$ **mod** 10.

Le espressioni sono importanti perché possono essere assegnate a una variabile, passate come argomento a una funzione (vedi più avanti), ed essere usate a loro volta in un'espressione. Espressioni sono:

- singole variabili (inclusi gli elementi di un array);
- singoli valori (inclusi quelli di tipo array);
- una chiamata a funzione (che vedremo più avanti);
- la composizione di due espressioni tramite un operatore: ad esempio, se espr1 ed espr2 sono espressioni, anche espr1 + espr2 è un'espressione;
- −(espr), dove espr è un'espressione che restituisce un intero, e **not** (espr), dove espr è un'espressione che restituisce vero o falso.

Per verificare se hai capito, puoi provare a individuare **tutte** le espressioni contenute nella seguente istruzione: $\mathsf{a} \leftarrow \mathsf{a} - \mathsf{b} \times 2$ (suggerimento: sono 6).

Strutture di controllo

Un programma composto da una semplice successione di istruzioni limita molto le possibilità. Considera il seguente pseudocodice:

Programma 3 Sequenza di istruzioni

```
1: variable a: integer
2: variable b: integer
3: variable c: integer
4: a ← 1
5: b ← 2
6: c ← 3
7: variable sum: integer
8: sum ← a + b + c
9: output(sum)
```

In questo frammento di pseudocodice, ogni riga è una singola istruzione. L'istruzione `output(sum)`, come vedremo tra poco, serve a stampare in output il valore di `sum`. Quello che il programma fa è sommare tre numeri "fissati" nel programma (1, 2 e 3): sostanzialmente qualcosa che avremmo potuto fare a mano. È in questo senso che è necessario introdurre complessità allo pseudocodice, poiché altrimenti non c'è molto che si possa fare. Le strutture di controllo hanno questo scopo (insieme alle funzioni, che vedremo nella sezione successiva).

Una **struttura di controllo** è un costrutto che racchiude un blocco di pseudocodice, detto *corpo*. A seconda del tipo di struttura di controllo, le istruzioni contenute nel corpo possono essere eseguite solo sotto particolari condizioni, oppure ripetute più volte. Le strutture di controllo possono essere annidate, ovvero il corpo di una struttura può contenerne delle altre. Il nostro pseudocodice utilizza tre tipi di strutture di controllo:

- Le strutture condizionali `if` e `if ... else`, con la seguente sintassi:

```
if {condizione} then
    {corpo if}
end if
```

```
if {condizione} then
    {corpo if}
else
    {corpo else}
end if
```

  Qui, {condizione} è un'espressione che restituisce vero o falso (ad esempio un confronto). Se è vera, viene eseguito {corpo if}. Per `if ... else`, qualora il valore di {condizione} sia falso, viene eseguito invece {corpo else}.
  Per esempio, questo programma assegna alla variabile `b` l'intero `100` se `a` vale `0`, e `101` se `a` vale `1` (negli altri casi non succede niente):

Programma 4 Uso di `if` e `if ... else`

```
1: variable b: integer
2: if a == 0 then
3:     b ← 100
4: else
5:     if a == 1 then
6:         b ← 101
7:     end if
8: end if
```

- Il ciclo `while`, con la seguente sintassi:

```
while {condizione} do
    {corpo}
end while
```

In questo caso, {corpo} viene ripetuto fintanto che {condizione} è vera. Quando il programma incontra un `while`, controlla prima se la condizione è vera: se non lo è, salta completamente il blocco e continua l'esecuzione. Se lo è, esegue una volta il corpo, e poi controlla nuovamente se la condizione è vera, ri-eseguendo il corpo in tal caso. Questo si ripete finché la condizione diventa falsa. Nota che questo ha senso se il valore di {condizione} dipende da quello che accade dentro {corpo}; altrimenti, è possibile che {condizione} sia sempre vera e il programma non esca mai dal ciclo: si parla in questo caso di *ciclo infinito*.
Ad esempio, questo programma calcola la somma dei multipli di 7 minori di n:

Programma 5 Uso di `while`

```
1: variable i: int
2: variable sum: int
3: i ← 0
4: sum ← 0
5: while i < n do
6:     sum ← sum + i
7:     i ← i + 7
8: end while
```

- Il ciclo `for`, con la seguente sintassi:

```
for {indice} in {intervallo} do
    {corpo}
end for
```

Vediamo cosa sono {indice} e {intervallo}. Partiamo dal secondo, che, come dice il nome, è un intervallo (di numeri interi), ovvero un insieme di numeri consecutivi. Per esempio, $\{-1, 0, 1, 2, 3\}$ e $\{7\}$ sono intervalli, mentre $\{3, 5\}$

non lo è. Dato che scrivere esplicitamente tutti i numeri di un intervallo è scomodo (e a volte impossibile), nello pseudocodice usiamo una notazione speciale per gli intervalli, ispirata da una tradizione matematica: `[`a ... b`)` indica l'intervallo di numeri che inizia da a e finisce in $b-1$ (incluso). Ovvero, l'estremo destro dell'intervallo è escluso. Quindi `[1 ... 3)` è l'intervallo $\{1, 2\}$ (si parla di "intervallo semiaperto"). Questa notazione può confondere all'inizio, ma il motivo di avere intervalli semiaperti è che rende più naturale iterare sugli array, ed è una prassi comune in molti linguaggi di programmazione.

Per quanto riguarda {indice}, esso è il nome di una nuova variabile temporanea che "scorre" sugli elementi dell'intervallo dal più piccolo al più grande. Chiamiamola `i` (spesso i nomi utilizzati per gli iteratori sono `i`, `j`, `k`, ...). È temporanea nel senso che esiste solo all'interno del `for`, e sparisce non appena il programma esce dal ciclo. Non va dichiarata (basta scriverne il nome), ed è sottinteso che sia di tipo *integer*. Il corpo del `for` viene eseguito un numero di volte pari alla lunghezza dell'intervallo (il numero di elementi che contiene). Durante la prima iterazione, `i` assume il valore a (l'estremo sinistro dell'intervallo). Durante la seconda iterazione, assume il valore $a+1$. E così via, fino all'ultima iterazione, durante la quale assume il valore $b-1$. La variabile `i` non viene mai modificata all'interno del corpo del ciclo.

Vediamo alcuni esempi:

Programma 6 `for` semplice

```
1: for i in [0 ... n) do
2:     output(i)
3: end for
```

Programma 7 Somma di array

```
1: variable sum: integer
2: sum ← 0
3: for i in [0 ... n) do
4:     sum ← sum + v[i]
5: end for
```

Programma 8 Ripetizione

```
1: variable equal_pair: integer
2: equal_pair ← 0
3: for i in [0 ... n) do
4:     for j in [i + 1 ... n) do
5:         if v[i] == v[j] then
6:             equal_pair ← 1
7:         end if
8:     end for
9: end for
```

Il programma 6 stampa in output i numeri da 0 a $n-1$. Il programma 7 calcola la somma degli elementi di un array v di lunghezza n. L'ultimo programma, 8, è un po' più complesso. Contiene due cicli `for` annidati. Il primo itera (tramite i) sugli elementi dell'array v, il secondo itera sugli elementi dell'array **che vengono dopo** i. Questo è il modo standard di "scandire" tutte le coppie di indici distinti. Quindi, alla fine dell'esecuzione, equal_pair vale 1 se e solo se è stata incontrata una coppia di elementi uguali, cioè una ripetizione.

Funzioni

Nei paragrafi precedenti abbiamo fatto spesso riferimento alle funzioni, senza mai chiarire cosa fossero. È finalmente arrivato il momento.

Una **funzione** è un blocco di pseudocodice, anche in questo caso chiamato *corpo*, racchiuso dalle parole chiave **function** e **end function**. Ci sono due motivazioni principali per l'utilizzo delle funzioni:

- permettere di riutilizzare parti di pseudocodice nel programma (a differenza dei cicli `while` e `for`, dove il corpo viene eseguito per più volte consecutive, una funzione può essere invocata, o chiamata, in punti arbirari del programma);
- rendere possibile la **ricorsione**, ovvero la capacità di una funzione di invocare se stessa.

Le componenti di una funzione sono:

- il **nome**: È ciò che identifica la funzione, e grazie al quale la si può invocare. Nel nostro pseudocodice, i nomi di funzioni sono formattati in blu.
- il **corpo**, di cui abbiamo già parlato: È la sequenza di istruzioni che viene eseguita quando la funzione viene chiamata.
- la lista di **parametri**: È una sequenza della forma var1: *tipo1*, var2: *tipo2*, ... che indica quanti valori, e di quali tipi, vanno "passati" come **argomenti** alla funzione quando questa viene chiamata. Una funzione può non avere parametri.
- il tipo del **valore di ritorno** (opzionale): Una funzione non può soltanto "fare" qualcosa, ma può anche *restituire* un valore — ad esempio, una funzione che somma due interi restituisce un intero. In questo caso, il tipo del valore di ritorno va indicato.

La sintassi di una funzione è la seguente (in alto senza valore di ritorno, in basso con valore di ritorno):

```
function fun(var1: tipo1, var2: tipo2, ...)
    {corpo}
end function
```

```
function fun(var1: tipo1, var2: tipo2, ...) → ritorno
    {corpo}
end function
```

Il nome della funzione è fun, tra parentesi tonde ci sono gli eventuali parametri (se non ce ne sono, si scrive semplicemente fun()), e *ritorno* è il tipo del valore di ritorno. Le entità var1, var2, ... sono variabili che possono essere usate solo dalla funzione, non dal codice esterno. I loro valori sono definiti solo al momento della chiamata a funzione, e vengono passati come argomenti (vedi sotto nel paragrafo dedicato). In particolare, possono essere diversi in chiamate diverse.

La parola chiave return

All'interno del corpo, è possibile usare la parola chiave **return** per restituire un valore (solo per le funzioni con valore di ritorno). Quando viene incontrato un **return**, l'esecuzione della funzione si interrompe e riprende dal punto in cui era stata chiamata; alla chiamata di funzione viene sostituito il valore che ha restituito.

Ad esempio, considera il seguente programma:

Programma 9 Funzioni e valore di ritorno

```
1: function add(a: integer, b: integer) → integer
2:     return a + b
3: end function
4: variable x: integer
5: variable y: integer
6: x ← -2
7: y ← 5
8: variable sum: integer
9: sum ← add(x, y)
```

La funzione add prende due parametri interi, a e b, e restituisce un intero. L'intero restituito corrisponde alla somma dei due parametri, come specificato dal **return** seguito dall'espressione a + b. Alla riga 10, la funzione viene chiamata dal programma, passando come argomenti le variabili x (che vale -2) e y (che vale 5). Pertanto, add restituisce $-2 + 5 = 3$, e questo valore viene assegnato a sum.

La parola chiave **return** può essere usata più di una volta nella stessa funzione. Per esempio, supponiamo di voler scrivere una funzione che restituisca il valore assoluto di un intero n (cioè, n stesso se $n \geq 0$, altrimenti $-n$). Potremmo farlo così:

Programma 10 Più di un **return**

```
1: function absolute_value(n: integer) → integer
2:     if n ≥ 0 then
3:         return n
4:     end if
5:     return −n
6: end function
```

Se ad absolute_value viene passato un valore ≥ 0, l'esecuzione del programma entra nel corpo dell'`if` e viene restituito n. In questo caso, l'esecuzione si interrompe: il programma "esce" immediatamente dalla funzione. In caso contrario, l'`if` viene saltato e viene eseguito il secondo **return**.

Ci si aspetterebbe che le funzioni senza tipo di ritorno non abbiano bisogno di alcun **return**. In effetti è così, ma a volte è comodo poter interrompere l'esecuzione di una funzione senza usare costrutti condizionali (che appesantiscono il codice). Per questo, si può usare un **return** "vuoto" (cioè non seguito da nulla) in un punto qualsiasi della funzione. Per esempio, questa funzione fa la stessa cosa di absolute_value nel programma 10, ma stampa il valore assoluto in output anziché restituirlo:

Programma 11 **return** vuoto

```
1: function print_absolute_value(n: integer)
2:     if n ≥ 0 then
3:         output(n)
4:         return
5:     end if
6:     output(−n)
7: end function
```

Chiamare una funzione

Se non fosse possibile invocare, o più comunemente "chiamare", una funzione, esse sarebbero inutili. Chiamare una funzione vuol dire spostare, temporaneamente, l'esecuzione del programma all'inizio della funzione stessa, "passando" una lista di argomenti, cioè i valori che assumono i parametri della funzione in quella chiamata. È importante chiarire la distinzione tra parametri e argomenti:

- i **parametri** sono le **variabili** che compaiono nella definizione della funzione, e, come tali, hanno un nome e un tipo;
- gli **argomenti** sono i **valori** (risultati di espressioni) che vengono passati alla funzione in una particolare chiamata.

Ovviamente, gli argomenti devono essere tanti quanti i parametri, e i loro tipi devono corrispondere. Abbiamo già visto un esempio di chiamata di funzione nel programma 9, ma anche tutte le volte in cui viene invocata `output`: questa, infatti, è una funzione senza tipo di ritorno con un unico parametro.

Funzioni ricorsive

La **ricorsione** è uno strumento molto potente nella programmazione, che non sarebbe possibile senza funzioni. Consiste nella possibilità di una funzione di chiamare se stessa, all'interno del proprio corpo. Quando ciò avviene, il programma ricomincia l'esecuzione della funzione con **nuovi argomenti** (quelli passati nella chiamata). Si tratta di un'esecuzione distinta da quella che l'ha invocata: quest'ultima resta "sospesa" finché la chiamata interna non termina, e poi riprende normalmente. I valori di tutte le variabili non vengono modificati. Una funzione che chiama se stessa almeno una volta si dice **funzione ricorsiva**.

Un esempio classico è il calcolo del fattoriale. Il fattoriale di un intero positivo n, indicato con $n!$, è il prodotto dei numeri tra 1 ed n, ovvero $1 \cdot 2 \cdots (n-1) \cdot n$. Si può calcolare facilmente usando le strutture di controllo (come un ciclo `for`), ma è istruttivo vedere come farlo tramite una funzione ricorsiva:

Programma 12 Funzione ricorsiva

```
1: function factorial(n: integer) → integer
2:     if n == 1 then
3:         return 1
4:     end if
5:     return n × factorial(n − 1)
6: end function
```

La funzione sfrutta il fatto che $n! = n \cdot (n-1)!$, come si vede alla riga 5. L'`if` alle righe 2-4 serve a fare in modo che la ricorsione si arresti: prima o poi, infatti, il valore di n diventerà 1, e a quel punto viene restituito 1 (il fattoriale di 1) senza ulteriori chiamate a factorial.

Funzioni di libreria

Il nostro pseudocodice mette a disposizione tre funzioni *di libreria*, ovvero disponibili senza dover esere definite: min, max e output. La funzione min prende come parametri due interi, e restituisce il minimo fra di essi. Analogamente, la funzione max prende come parametri due interi, e restituisce il massimo fra di essi. Ad esempio, chiamando min(-3, 2) viene restituito -3, e chiamando max(a, a + 1) viene restituito in ogni caso il valore di a + 1. La funzione output può essere chiamata con un argomento di tipo intero per stamparlo in output, oppure può essere chiamata con la forma output("pippo") che stampa la parola *pippo* in output.

Scambio di variabili

Un'operazione spesso utile in programmazione è scambiare i valori di due variabili. Come farlo è meno ovvio di quanto sembri: per scambiare i valori di a e b, non possiamo semplicemente scrivere in sequenza gli assegnamenti a ← b e b ← a. Infatti, subito dopo il primo assegnamento entrambe le variabili avranno lo stesso valore di b: il valore di a è andato perso! Un modo per ovviare è introdurre una terza variabile "ausiliaria", chiamiamola ad esempio x, in questo modo:

```
x ← a
a ← b
b ← x
```

In pratica, x serve a memorizzare il valore di a prima che venga sovrascritto da quello di b, così che possa poi essere assegnato a b stesso.

Dato che, come già detto, lo scambio è molto frequente, nello pseudocodice usiamo una espressione dedicata esclusivamente a questo scopo.

```
(a, b) ← (b, a)
```

L'espressione `(a, b)` può essere usata **esclusivamente** in questo contesto. Le variabili che compaiono a destra devono essere le stesse che compaiono a sinistra, ma in ordine invertito. Questa istruzione compie esattamente lo scambio delle due variabili: è come se fosse un "assegnamento simultaneo".

Esempi di pseudocodice

Vediamo ora altri esempi concreti, che valgono più di mille parole.

Implementazioni di algoritmi classici

In questa sezione trovi una lista di brevi programmi in pseudocodice che implementano algoritmi più o meno semplici e di natura didattica. Sono tutti proposti sotto forma di funzione. Sono presenti descrizioni molto minimali: una spiegazione esaustiva dello pseudocodice è stata volutamente omessa, per incentivarti a comprendere autonomamente il funzionamento del programma.

Capire se un numero è primo

La funzione is_prime restituisce 1 se n è un numero primo, e 0 altrimenti.

Programma 13 Test di primalità

```
1:  function is_prime(n: integer) → integer
2:      variable i: integer
3:      i ← 2
4:      while i × i ≤ n do
5:          if n mod i == 0 then
6:              return 0
7:          end if
8:          i ← i + 1
9:      end while
10:     return 1
11: end function
```

Stampare tutti i primi fino a 1000

La funzione print_primes si avvale della funzione is_prime dell'esempio precedente per stampare, in ordine, tutti i numeri primi compresi tra 1 e 1000.

Programma 14 Stampa i primi ≤ 1000

```
1: function print_primes()
2:     for i in [1 ... 1001) do
3:         if is_prime(i) == 1 then
4:             output(i)
5:         end if
6:     end for
7: end function
```

Ribaltare un array

La funzione reverse inverte gli elementi di un array v di lunghezza n.

Programma 15 Ribalta un array

```
1: function reverse(n: integer, v: integer[]) → integer[]
2:     for i in [0 ... n / 2) do
3:         variable j: integer
4:         j ← n − 1 − i
5:         (v[i], v[j]) ← (v[j], v[i])
6:     end for
7:     return v
8: end function
```

Massima somma di un prefisso

Un *prefisso* di un array è una sottosequenza contigua di elementi che parte da quello di indice 0. La funzione max_prefix trova la massima somma di un prefisso di un array v di lunghezza n.

Programma 16 Prefisso di somma massima

```
1:  function max_prefix(n: integer, v: integer[]) → integer
2:      variable maximum: integer
3:      variable sum: integer
4:      maximum ← v[0]
5:      sum ← 0
6:      for i in [0 ... n) do
7:          sum ← sum + v[i]
8:          if sum > maximum then
9:              maximum ← sum
10:         end if
11:     end for
12:     return maximum
13: end function
```

Contare il numero di somme uguali a un numero dato

La funzione ricorsiva count_sums restituisce il numero di modi di scrivere un intero positivo n come somma **ordinata** di addendi positivi. Per esempio, $n = 4$ si scrive in 8 modi: $1+1+1+1$, $1+1+2$, $1+2+1$, $2+1+1$, $2+2$, $1+3$, $3+1$, 4 (si considera anche la somma composta dal solo addendo n).

Programma 17 Numero di somme uguali a n

```
1:  function count_sums(n: integer) → integer
2:      if n == 1 then
3:          return 1
4:      end if
5:      variable answer: integer
6:      answer ← 1
7:      for i in [1 ... n) do
8:          answer ← answer + count_sums(i)
9:      end for
10:     return answer
11: end function
```

Somma di una sottosequenza

Questo programma è più complesso dei precedenti.

Una *sottosequenza* di un array è un sottoinsieme dei suoi elementi (anche non consecutivi). Dati un array di interi `v` di lunghezza `n` e un intero `x`, la chiamata `subsequence_sum(n, v, x, 0)` ritorna `1` se esiste una sottosequenza di `v` con somma `x`, e `0` altrimenti. Per convenzione, la sottosequenza vuota ha somma `0`.

Programma 18 Esiste una sottosequenza di somma x?

```
1:  function subsequence_sum(n: integer, v: integer[], x: integer, sum: integer) → integer
2:      if n == 0 then
3:          if sum == x then
4:              return 1
5:          end if
6:          return 0
7:      end if
8:      if subsequence_sum(n − 1, v, x, sum) == 1 then
9:          return 1
10:     end if
11:     if subsequence_sum(n − 1, v, x, sum + v[n − 1]) == 1 then
12:         return 1
13:     end if
14:     return 0
15: end function
```

Esempi da prove passate

Seguono degli esercizi tratti dalle ultime due edizioni della selezione scolastica. In fondo trovi le risposte a tutti gli esercizi, con una breve spiegazione per ciascuna di esse.

Selezioni scolastiche 2018-19, Esercizio 6

Calcolare per quante volte viene eseguito il ciclo `while` nel seguente pseudocodice:

Programma 19 2018-19, Esercizio 6

```
1:  variable conta: integer
2:  variable alfa: integer
3:  variable beta: integer
4:  conta ← 0
5:  alfa ← 0
6:  beta ← 0
7:  while conta < 29 do
8:      if conta mod 3 == 1 then
9:          alfa ← alfa + 2
10:     else
11:         beta ← beta + 1
12:     end if
13:     conta ← conta + 2
14: end while
```

Selezioni scolastiche 2018-19, Esercizio 9

Sono date le seguenti funzioni:

Programma 20 2018-19, Esercizio 9

```
1:  function mystery(a: integer, b: integer) → integer
2:      if 2 × a > b then
3:          return b
4:      else
5:          return a
6:      end if
7:  end function
8:
9:  function secret(a: integer, b: integer) → integer
10:     if a + b > mystery(a, b) then
11:         return a
12:     else
13:         return mystery(a, b)
14:     end if
15: end function
```

Quale valore viene restituito dalla chiamata `secret(24, 3)`?

Selezioni scolastiche 2020-21, Esercizio 6

Nel seguente programma, qual è il valore dell'ultimo intero che viene stampato durante l'esecuzione?

Programma 21 2020-21, Esercizio 6

```
1:  variable v: integer[]
2:  variable w: integer[]
3:  v ← [4, 2, 6, 3, 5, 8, 9, 0, 7, 1]
4:  w ← [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
5:  for i in [0 ... 10) do
6:      w[v[i]] ← i
7:  end for
8:  for i in [0 ... 10) do
9:      output(w[i])
10: end for
```

Selezioni scolastiche 2020-21, Esercizio 7

È data la seguente funzione:

Programma 22 2020-21, Esercizio 7

```
1:  function f(x: integer) → integer
2:      variable i: integer
3:      while x > 0 do
4:          if x mod 2 == 0 then
5:              x ← x / 2
6:          else
7:              x ← x − 1
8:          end if
9:          i ← i + 1
10:     end while
11:     return i
12: end function
```

Qual è il valore minimo da passare a `x` affinché `f` ritorni 5?

Selezioni scolastiche 2021-22, Esercizio 6

È data la seguente funzione:

Programma 23 2021-22, Esercizio 6

```
1: function f(n: integer) → integer
2:     if n < 10 then
3:         return f(n + 1) + 3
4:     else
5:         if n == 10 then
6:             return 7
7:         else
8:             return f(n − 2) − 1
9:         end if
10:    end if
11: end function
```

Quanto vale `f(13)`?

Soluzioni

Selezioni 2018-19, Esercizio 6. La risposta è $\boxed{15}$.
Tutto quello che succede nel corpo del `while` è irrilevante, perché non influenza il valore di `conta`! Quest'ultimo aumenta di 2 a ogni iterazione del ciclo. All'inizio vale 0, e il programma esce dal ciclo quando raggiunge il valore 30. Quindi il `while` viene eseguito $30/2 = 15$ volte.

Selezioni 2018-19, Esercizio 9. La risposta è $\boxed{24}$.
Si tratta di simulare il programma: la chiamata `mystery(a, b)`, con `a` e `b` che valgono, rispettivamente, 24 e 3, restituisce `b`, cioè 3, perché $2 \cdot 24 > 3$. Allora la condizione `a + b > mystery(a, b)` è vera ($24 + 3 > 3$), per cui viene `mystery` restituisce il valore di `a`, ovvero 24.

Selezioni 2020-21, Esercizio 6. La risposta è $\boxed{6}$.
L'ultimo intero a essere stampato è `w[9]`. Quindi, nel `for` alle righe 5-7 importa solo l'iterazione in cui `v[i]` assume il valore 9. Questo succede quando `i` vale 6, e l'istruzione `w[v[i]] ← i` assegna tale valore a `w[9]`.

Selezioni 2020-21, Esercizio 7. La risposta è $\boxed{7}$.
Andiamo a ritroso. Il `while` si arresta quando `x` vale 0. All'iterazione precedente, `x` non poteva che valere $0 + 1 = 1$ (non può essere stato dimezzato perché allora sarebbe stato 0 già prima). All'iterazione ancora precedente, `x` valeva $2 \cdot 1 = 2$. Proseguendo, prima ancora `x` poteva valere o $2 + 1 = 3$, oppure $2 \cdot 2 = 4$: siccome vogliamo il minimo, conviene scegliere la prima opzione. Subito prima `x` valeva per forza $2 \cdot 3 = 6$ (non poteva valere 4, altrimenti sarebbe stato dimezzato). Infine, all'iterazione precedente (siamo arrivati a 5) `x` valeva $6 + 1 = 7$ oppure $2 \cdot 6 = 12$.

Selezioni 2021-22, Esercizio 6. La risposta è $\boxed{8}$.
La funzione è ricorsiva. Quando viene chiamata `f(13)`, siamo nel terzo caso (13 non è né minore né uguale a 10). Allora viene restituito `f(13 − 2) − 1`, quindi dobbiamo capire cosa fa `f(11)`. Ricadiamo ancora nel terzo caso, quindi viene restituito `f(9) − 1`, ovvero la chiamata iniziale restituisce `f(9) − 2`. Quando viene chiamata `f(9)`, siamo nel primo caso ($9 < 10$), e viene restituito `f(9 + 1) + 3`, che, per la chiamata iniziale, dà `f(10) + 1`. Dato che `f(10)` resituisce 7, la risposta è $7 + 1 = 8$.