Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Lo scopo di questi esempi è solo didattico, utilizzando forme non ottimizzate per la velocità di esecuzione.

Problemi elementari di programmazione 1079
Somma tra due numeri positivi 1079
Moltiplicazione di due numeri positivi attraverso la somma 1080
Divisione intera tra due numeri positivi 1081
Elevamento a potenza 1081
Radice quadrata 1082
Fattoriale 1083
Massimo comune divisore 1083
Numero primo 1084
Scansione di array 1084
Ricerca sequenziale 1084
Ricerca binaria 1085
Algoritmi tradizionali 1086
Bubblesort 1086
Torre di Hanoi 1087
Quicksort 1087
Permutazioni 1089

Problemi elementari di programmazione

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Scheme.

Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è descritto nella sezione 62.3.1.

```
; ======================================================================
; somma1.scm
; Somma esclusivamente valori positivi.
; ======================================================================

; ======================================================================
; (somma <x> <y>)
; ----------------------------------------------------------------------
(define (somma x y)
    (define z x)
```

```
    (define i 1)

    (do ()
        ((> i y))

        (set! z (+ z 1))
        (set! i (+ i 1))
    )

    z
)

; ================================================================
; Inizio del programma.
; ----------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (somma x y))
(display x) (display " + ") (display y) (display " = ") (display z)
(newline)

; ================================================================
```

In alternativa, si può modificare la funzione '**somma**', in modo che il ciclo '**do**' gestisca la dichiarazione e l'incremento delle variabili che usa. Tuttavia, in questo caso, la variabile '**z**' deve essere «copiata» in modo da poter trasmettere il risultato all'esterno del ciclo '**do**':

```
(define (somma x y)
    (define risultato 0)

    (do ((z x (+ z 1)) (i 1 (+ i 1)))
        ((> i y))
        (set! risultato z)
    )

    risultato
)
```

Volendo gestire la cosa in modo un po' più elegante, occorre togliere la variabile '**z**' dalla gestione del ciclo '**do**':

```
(define (somma x y)
    (define z x)

    (do ((i 1 (+ i 1)))
        ((> i y))
        (set! z (+ z 1))
    )

    z
)
```

Moltiplicazione di due numeri positivi attraverso la somma

«

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è descritto nella sezione 62.3.2.

```
; ================================================================
; moltiplica1.scm
; Moltiplica esclusivamente valori positivi.
; ================================================================

; ================================================================
; (moltiplica <x> <y>)
; ----------------------------------------------------------------
(define (moltiplica x y)
    (define z 0)
    (define i 1)

    (do ()
        ((> i y))

        (set! z (+ z x))
        (set! i (+ i 1))
    )

    z
)

; ================================================================
; Inizio del programma.
; ----------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
```



```
(set! z (moltiplica x y))
(display x) (display " * ") (display y) (display " = ") (display z)
(newline)

; ================================================================
```

In alternativa, si può modificare la funzione '**moltiplica**', in modo che il ciclo '**do**' gestisca la dichiarazione e l'incremento dell'indice '**i**':

```
(define (moltiplica x y)
    (define z 0)

    (do ((i 1 (+ i 1)))
        ((> i y))

        (set! z (+ z x))
    )

    z
)
```

Divisione intera tra due numeri positivi

«

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è descritto nella sezione 62.3.3.

```
; ================================================================
; dividi1.scm
; Divide esclusivamente valori positivi.
; ================================================================

; ================================================================
; (dividi <x> <y>)
; ----------------------------------------------------------------
(define (dividi x y)
    (define z 0)
    (define i x)

    (do ()
        ((< i y))

        (set! i (- i y))
        (set! z (+ z 1))
    )

    z
)

; ================================================================
; Inizio del programma.
; ----------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (dividi x y))
(display x) (display " / ") (display y) (display " = ") (display z)
(newline)

; ================================================================
```

In alternativa, si può modificare la funzione '**dividi**', in modo che il ciclo '**do**' gestisca la dichiarazione e il decremento della variabile '**i**'. Per la precisione, la variabile '**z**' non può essere dichiarata nello stesso modo, perché serve anche al di fuori del ciclo:

```
(define (dividi x y)
    (define z 0)

    (do ((i x (- i y)))
        ((< i y))

        (set! z (+ z 1))
    )

    z
)
```

Elevamento a potenza

«

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è descritto nella sezione 62.3.4.

```
; ================================================================
; potenza1.scm
; Eleva a potenza.
; ================================================================

; ================================================================
; (potenza <x> <y>)
; ----------------------------------------------------------------
```

```
(define (potenza x y)
    (define z 1)
    (define i 1)

    (do ()
        ((> i y))

        (set! z (* z x))
        (set! i (+ i 1))
    )

    z
)

; ====================================================================
; Inizio del programma.
; --------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (potenza x y))
(display x) (display " ** ") (display y) (display " = ") (display z)
(newline)

; ====================================================================
```

In alternativa, si può modificare la funzione '**`potenza`**', in modo che il ciclo '**`do`**' gestisca la dichiarazione e l'incremento della variabile '**`i`**':

```
(define (potenza x y)
    (define z 1)

    (do ((i 1 (+ i 1)))
        ((> i y))

        (set! z (* z x))
    )

    z
)
```

È possibile usare anche un algoritmo ricorsivo:

```
(define (potenza x y)
    (if (= x 0)
        0
        (if (= y 0)
            1
            (* x (potenza x (- y 1)))
        )
    )
)
```

Radice quadrata

Il problema della radice quadrata è descritto nella sezione 62.3.5.

```
; ====================================================================
; radice1.scm
; Radice quadrata.
; ====================================================================

; ====================================================================
; (radice <x>)
; --------------------------------------------------------------------
(define (radice x)
    (define z -1)
    (define t 0)
    (define uscita #f)

    (do ()
        (uscita)

        (set! z (+ z 1))
        (set! t (* z z))
        (if (> t x)
            ; È stato superato il valore massimo
            (begin
                (set! z (- z 1))
                (set! uscita #t)
            )
        )
    )

    z
)

; ====================================================================
; Inizio del programma.
; --------------------------------------------------------------------
(define x 0)
(define z 0)

(display "Inserisci il numero intero positivo: ")
```



```
(set! x (read))
(newline)
(set! z (radice x))
(display "La radice quadrata di ") (display x) (display " è ") (display z)
(newline)

; ====================================================================
```

Fattoriale

Il problema del fattoriale è descritto nella sezione 62.3.6.

```
; ====================================================================
; fattoriale1.scm
; Fattoriale.
; ====================================================================

; ====================================================================
; (fattoriale <x>)
; --------------------------------------------------------------------
(define (fattoriale x)
    (define i (- x 1))

    (do ()
        ((<= i 0))

        (set! x (* x i))
        (set! i (- i 1))
    )

    x
)

; ====================================================================
; Inizio del programma.
; --------------------------------------------------------------------
(define x 0)
(define z 0)

(display "Inserisci il numero intero positivo: ")
(set! x (read))
(newline)
(set! z (fattoriale x))
(display x) (display "! = ") (display z)
(newline)

; ====================================================================
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo:

```
(define (fattoriale x)
    (if (> x 1)
        (* x (fattoriale (- x 1)))
        1
    )
)
```

Massimo comune divisore

Il problema del massimo comune divisore, tra due numeri positivi, è descritto nella sezione 62.3.7.

```
; ====================================================================
; mcd1.scm
; Massimo Comune Divisore.
; ====================================================================

; ====================================================================
; (moltiplica <x> <y>)
; --------------------------------------------------------------------
(define (mcd x y)
    (do ()
        ((= x y))

        (if (> x y)
            (set! x (- x y))
            (set! y (- y x))
        )
    )

    x
)

; ====================================================================
; Inizio del programma.
; --------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (mcd x y))
(display "MCD di ") (display x) (display " e ") (display y)
(display " è ") (display z)
(newline)
```

```
; =====================================================================
```

Numero primo

Il problema della determinazione se un numero sia primo o meno, è descritto nella sezione 62.3.8.

```
; =====================================================================
; primo1.scm
; Numero primo.
; =====================================================================

; =====================================================================
; (primo <x>)
; ---------------------------------------------------------------------
(define (primo x)
    (define np #t)
    (define i 2)
    (define j 0)

    (do ()
        ((or (>= i x) (not np)))

        (set! j (truncate (/ x i)))
        (set! j (- x (* j i)))
        (if (= j 0)
            (set! np #f)
            (set! i (+ i 1))
        )
    )

    np
)

; =====================================================================
; Inizio del programma.
; ---------------------------------------------------------------------
(define x 0)

(display "Inserisci un numero intero positivo: ")
(set! x (read))
(newline)
(if (primo x)
    (display "È un numero primo")
    (display "Non è un numero primo")
)
(newline)

; =====================================================================
```

Scansione di array

Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è descritto nella sezione 62.4.1.

```
; =====================================================================
; ricerca_sequenziale1.scm
; Ricerca Sequenziale.
; =====================================================================

; =====================================================================
; (ricerca <vettore> <x> <ele-inf> <ele-sup>)
; ---------------------------------------------------------------------
(define (ricerca vettore x a z)
    (define risultato -1)

    (do ((i a (+ i 1)))
        ((> i z))

        (if (= x (vector-ref vettore i))
            (set! risultato i)
        )
    )
    risultato
)

; =====================================================================
; Inizio del programma.
; ---------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
```



```
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(display "Inserire il valore da cercare: ")
(set! x (read))
(newline)

(set! i (ricerca vettore x 0 (- z 1)))

(display "Il valore cercato si trova nell'elemento ")
(display i)
(newline)

; =====================================================================
```

Esiste anche una soluzione ricorsiva che viene mostrata di seguito:

```
(define (ricerca vettore x a z)
    (if (> a z)
        ; La corrispondenza non è stata trovata.
        1
        (if (= x (vector-ref vettore a))
            a
            (ricerca vettore x (+ a 1) z)
        )
    )
)
```

Ricerca binaria

Il problema della ricerca binaria all'interno di un array, è descritto nella sezione 62.4.2.

```
; =====================================================================
; ricerca_binaria1.scm
; Ricerca Binaria.
; =====================================================================

; =====================================================================
; (ricerca <vettore> <x> <ele-inf> <ele-sup>)
; ---------------------------------------------------------------------
(define (ricerca vettore x a z)
    (define m (truncate (/ (+ a z) 2)))

    (if (or (< m a) (> m z))
        ; Non restano elementi da controllare: l'elemento cercato
        ; non c'è.
        -1

        (if (< x (vector-ref vettore m))
            ; Si ripete la ricerca nella parte inferiore.
            (ricerca vettore x a (- m 1))

            (if (> x (vector-ref vettore m))
                ; Si ripete la ricerca nella parte superiore.
                (ricerca vettore x (+ m 1) z)

                ; Se x è uguale a vettore[m], l'obiettivo è
                ; stato trovato.
                m
            )
        )
    )
)

; =====================================================================
; Inizio del programma.
; ---------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore (in modo ordinato).")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
```

```
)

(display "Inserire il valore da cercare: ")
(set! x (read))
(newline)

(set! i (ricerca vettore x 0 (- z 1)))

(display "Il valore cercato si trova nell'elemento ")
(display i)
(newline)

; ================================================================
```

Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Scheme.

Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 62.5.1. Viene mostrato prima una soluzione iterativa e successivamente la funzione '**bsort**' in versione ricorsiva.

```
; ================================================================
; bsort1.scm
; Bubblesort.
; ================================================================

; ================================================================
; (ordina <vettore> <ele-inf> <ele-sup>)
; ----------------------------------------------------------------
(define (ordina vettore a z)
    (define scambio 0)

    (do ((j a (+ j 1)))
        ((>= j z))

        (do ((k (+ j 1) (+ k 1)))
            ((> k z))

            (if (< (vector-ref vettore k) (vector-ref vettore j))
                ; Scambia i valori.
                (begin
                    (set! scambio (vector-ref vettore k))
                    (vector-set! vettore k (vector-ref vettore j))
                    (vector-set! vettore j scambio)
                )
            )
        )
    )

    vettore
)

; ================================================================
; Inizio del programma.
; ----------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(set! vettore (ordina vettore 0 (- z 1)))

(display "Il vettore ordinato è il seguente: ")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display (vector-ref vettore i))
    (display " ")
)
(newline)

; ================================================================
```


Segue la funzione '**ordina**' in versione ricorsiva:

```
(define (ordina vettore a z)
    (define scambio 0)

    (if (< a z)
        (begin
            ; Scansione interna dell'array per collocare nella
            ; posizione a l'elemento giusto.
            (do ((k (+ a 1) (+ k 1)))
                ((> k z))

                (if (< (vector-ref vettore k) (vector-ref vettore a))
                    ; Scambia i valori.
                    (begin
                        (set! scambio (vector-ref vettore k))
                        (vector-set! vettore k (vector-ref vettore a))
                        (vector-set! vettore a scambio)
                    )
                )
            )

            (set! vettore (ordina vettore (+ a 1) z))
        )
    )
    vettore
)
```

Torre di Hanoi

Il problema della torre di Hanoi è descritto nella sezione 62.5.3.

```
; ================================================================
; hanoi1.scm
; Torre di Hanoi.
; ================================================================

; ================================================================
; (hanoi <n-anelli> <piolo-iniziale> <piolo-finale>)
; ----------------------------------------------------------------
(define (hanoi n p1 p2)
    (if (> n 0)
        (begin
            (hanoi (- n 1) p1 (- 6 (+ p1 p2)))
            (begin
                (display "Muovi l'anello ")
                (display n)
                (display " dal piolo ")
                (display p1)
                (display " ")
                (display p2)
                (newline)
            )
            (hanoi (- n 1) (- 6 (+ p1 p2)) p2)
        )
    )
)


; ================================================================
; Inizio del programma.
; ----------------------------------------------------------------
(define n 0)
(define p1 0)
(define p2 0)

(display "Inserisci il numero di pioli: ")
(set! n (read))
(newline)
(display "Inserisci il numero del piolo iniziale (da 1 a 3): ")
(set! p1 (read))
(newline)
(display "Inserisci il numero del piolo finale (da 1 a 3): ")
(set! p2 (read))
(newline)
(hanoi n p1 p2)

; ================================================================
```

Quicksort

L'algoritmo del Quicksort è stato descritto nella sezione 62.5.4.

```
; ================================================================
; qsort1.scm
; Quicksort.
; ================================================================

; ----------------------------------------------------------------
; Dichiara il vettore a cui successivamente fanno riferimento tutte le
; funzioni.
; Il vettore non viene passato alle funzioni tra gli argomenti, per
; semplificare le funzioni, soprattutto nel caso di «part», che
; deve restituire anche un altro valore.
; ----------------------------------------------------------------
(define DIM 100)
(define vettore (make-vector DIM))

; ================================================================
; (inverti-elementi <indice-1> <indice-2>)
; ----------------------------------------------------------------
(define (inverti-elementi a z)
```

```
    (define scambio 0)
    (set! scambio (vector-ref vettore a))
    (vector-set! vettore a (vector-ref vettore z))
    (vector-set! vettore z scambio)
)

; ====================================================================
; (part <ele-inf> <ele-sup>)
; --------------------------------------------------------------------
(define (part a z)
    ; Si assume che «a» sia inferiore a «z».
    (define i (+ a 1))
    (define cf z)
    ; Vengono preparate delle variabili per controllare l'uscita dai cicli.
    (define uscita1 #f)
    (define uscita2 #f)
    (define uscita3 #f)

    ; Inizia il ciclo di scansione dell'array.
    (set! uscita1 #f)
    (do ()
        (uscita1)
        (set! uscita2 #f)
        (do ()
            (uscita2)

            ; Sposta «i» a destra.
            (if (or
                    (> (vector-ref vettore i) (vector-ref vettore a))
                    (>= i cf)
                )
                ; Interrompe il ciclo interno.
                (set! uscita2 #t)
                ; Altrimenti incrementa l'indice
                (set! i (+ i 1))
            )
        )
        (set! uscita3 #f)
        (do ()
            (uscita3)

            ; Sposta «cf» a sinistra.
            (if (<= (vector-ref vettore cf) (vector-ref vettore a))
                ; Interrompe il ciclo interno.
                (set! uscita3 #t)
                ; Altrimenti decrementa l'indice
                (set! cf (- cf 1))
            )
        )

        (if (<= cf i)
            ; È avvenuto l'incontro tra «i» e «cf».
            (set! uscita1 #t)
            ; Altrimenti vengono scambiati i valori.
            (begin
                (inverti-elementi i cf)
                (set! i (+ i 1))
                (set! cf (- cf 1))
            )
        )
    )

    ; A questo punto vettore[a..z] è stato ripartito e «cf» è la
    ; collocazione di vettore[a].
    (inverti-elementi a cf)

    ; A questo punto, vettore[cf] è un elemento (un valore) nella
    ; posizione giusta, e «cf» è ciò che viene restituito.
    cf
)

; ====================================================================
; (ordina <ele-inf> <ele-sup>)
; --------------------------------------------------------------------
(define (ordina a z)
    ; Viene preparata la variabile «cf».
    (define cf 0)

    (if (> z a)
        (begin
            (set! cf (part a z))
            (ordina a (- cf 1))
            (ordina (+ cf 1) z)
        )
    )
)

; ====================================================================
; Inizio del programma.
; --------------------------------------------------------------------

(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

; Il vettore non viene trasferito come argomento della funzione,
; ma risulta accessibile esternamente.
(ordina 0 (- z 1))

(display "Il vettore ordinato è il seguente: ")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display (vector-ref vettore i))
    (display " ")
)
(newline)

; ====================================================================
```


Permutazioni

L'algoritmo ricorsivo delle permutazioni è descritto nella sezione 62.5.5.

```
; ====================================================================
; permuta1.scm
; Permutazioni.
; ====================================================================

; --------------------------------------------------------------------
; Dichiara il vettore a cui successivamente fanno riferimento tutte le
; funzioni.
; --------------------------------------------------------------------
(define DIM 100)
(define vettore (make-vector DIM))

; --------------------------------------------------------------------
; Sempre per motivi pratici, rende disponibile la dimensione utilizzata
; effettivamente.
; --------------------------------------------------------------------
(define n-elementi 0)

; ====================================================================
; (inverti-elementi <indice-1> <indice-2>)
; --------------------------------------------------------------------
(define (inverti-elementi a z)
    (define scambio 0)
    (set! scambio (vector-ref vettore a))
    (vector-set! vettore a (vector-ref vettore z))
    (vector-set! vettore z scambio)
)

; ====================================================================
; (visualizza)
; --------------------------------------------------------------------
(define (visualizza)
    (do ((i 0 (+ i 1)))
        ((>= i n-elementi))

        (display (vector-ref vettore i))
        (display " ")
    )
    (newline)
)

; ====================================================================
; (permuta <inizio> <fine>)
; --------------------------------------------------------------------
(define (permuta a z)
    (define k 0)

    ; Se il segmento di array contiene almeno due elementi, si
    ; procede.
    (if (>= (- z a) 1)
        ; Inizia un ciclo di scambi tra l'ultimo elemento e uno
        ; degli altri contenuti nel segmento di array.
        (do ((k z (- k 1)))
            ((< k a))

            ; Scambia i valori.
            (inverti-elementi k z)

            ; Esegue una chiamata ricorsiva per permutare un
            ; segmento più piccolo dell'array.
            (permuta a (- z 1))

            ; Scambia i valori.
            (inverti-elementi k z)
        )

        ; Altrimenti, visualizza l'array e utilizza una variabile
```

```
        ; dichiarata globalmente.
        (visualizza)
    )
)

; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! n-elementi (read))
(newline)

(if (> n-elementi DIM)
    (set! n-elementi DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i n-elementi))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

; Il vettore non viene trasferito come argomento della funzione,
; ma risulta accessibile esternamente.
(permuta 0 (- n-elementi 1))

; ======================================================================
```