Capitolo 62

Algoritmi e notazioni

62.1 Notazione BNF 17
62.1.1 BNF essenziale 17
62.1.2 Estensioni usuali 18
62.2 Pseudocodifica 20
62.3 Problemi elementari di programmazione 22
62.3.1 Somma tra due numeri positivi 22
62.3.2 Moltiplicazione di due numeri positivi attraverso la somma 24
62.3.3 Divisione intera tra due numeri positivi 25
62.3.4 Elevamento a potenza 26
62.3.5 Radice quadrata 28
62.3.6 Fattoriale 29
62.3.7 Massimo comune divisore 30
62.3.8 Numero primo 31
62.3.9 Successione di Fibonacci 32
62.4 Scansione di array 34
62.4.1 Ricerca sequenziale 34
62.4.2 Ricerca binaria 36
62.5 Problemi classici di programmazione 37
62.5.1 Bubblesort 37
62.5.2 Fusione tra due array ordinati 39
62.5.3 Torre di Hanoi 41

62.5.4 Quicksort (ordinamento non decrescente) 44
62.5.5 Permutazioni 51
62.6 Gestione dei file 53
62.6.1 Fusione tra due file ordinati 53
62.6.2 Riordino attraverso la fusione 56
62.7 Trasformazione in lettere 66
62.7.1 Da numero a sequenza alfabetica pura 66
62.7.2 Da numero a numero romano 69
62.7.3 Da numero a lettere, nel senso verbale 75
62.8 Algoritmi elementari con la shell POSIX 79
62.8.1 ARCS0: ricerca del valore più grande tra tre numeri interi 80
62.8.2 ARCS1: moltiplicazione di due numeri interi 81
62.8.3 ARCS2: valore assoluto della differenza tra due valori 82
62.8.4 ARCS3: somma tra due numeri 83
62.8.5 ARCS4: prodotto tra due numeri 83
62.8.6 ARCS5: quoziente 85
62.8.7 ARCS6: verifica della parità di un numero 86
62.8.8 ARCS7: fattoriale 87
62.8.9 ARCS8: coefficiente binomiale 87
62.8.10 ARCS10: massimo comune divisore 89
62.8.11 ARCS11: massimo comune divisore 90
62.8.12 ARCS12: radice quadrata intera 91

62.8.13 ARCS13: numero primo 92
62.8.14 ARCS14: numero primo 93
62.9 Riferimenti 94

62.1 Notazione BNF

In molti documenti si usa la «notazione BNF» per mostrare la sintassi di qualcosa, particolarmente quando si tratta della descrizione formale dei linguaggi di programmazione. La sigla BNF sta per *Backus Naur form*, a ricordare che si tratta di una notazione introdotta da John Backus e Peter Naur, tra il 1959 e il 1960.

62.1.1 BNF essenziale

La notazione BNF utilizza pochi simboli per attribuire un significato a ciò che descrive:

Simbolo	Significato
::=	Si legge come: «è definito da». Sta a indicare che l'oggetto alla sinistra di tale simbolo viene definito come ciò che si trova alla destra di questo.
|	Si legge come: «oppure». Sta a indicare che può essere usato l'oggetto che sta a sinistra del simbolo, oppure quello a destra, indifferentemente.
<***nome***>	Indica il nome di una categoria. Il nome può essere scritto senza vincoli particolari.
x	Qualunque cosa sia scritta al di fuori delle parentesi angolari ('**<**', '**>**'), escludendo altri simboli di cui sia stato dichiarato il significato, va interpretata letteralmente (come parola chiave).

A titolo di esempio, viene mostrata la definizione di una lettera dell'alfabeto latino, suddividendo il problema, definendo cosa sono le lettere latine minuscole e cosa sono le lettere latine maiuscole:

```
<lettera_alfabeto_latino> ::=
        <lettera_alfabeto_latino_maiuscola>
      | <lettera_alfabeto_latino_minuscola>
```

```
<lettera_alfabeto_latino_maiuscola> ::=
        A | B | C | D | E | F | G | H | I | J | K | L
      | M | N | O | P | Q | R | S | T | U | V | W | X
      | Y | Z
```

```
<lettera_alfabeto_latino_minuscola> ::=
        a | b | c | d | e | f | g | h | i | j | k | l
      | m | n | o | p | q | r | s | t | u | v | w | x
      | y | z
```

62.1.2 Estensioni usuali

«

Di fatto, la notazione BNF viene usata estendendo la simbologia, per semplificarne la lettura ed evitare ambiguità, ma a volte la simbologia viene anche cambiata leggermente.

Simbolo	Significato
`[...]`	Le parentesi quadre vengono usate normalmente per delimitare una porzione facoltativa della dichiarazione.

Simbolo	Significato
{...}	Le parentesi graffe vengono usate normalmente per delimitare una porzione necessaria della dichiarazione, che però, per qualche ragione, va intesa come un blocco unito. A volte, l'inclusione tra parentesi graffe va intesa come la possibilità di ripetere indefinitamente il suo contenuto, ma per questo, di solito si aggiungono tre puntini di sospensione in coda.
. . .	Tre puntini di sospensione vengono usati per indicare una porzione della dichiarazione che può essere ripetuta.
"..." '...'	Una coppia di apici doppi o singoli, può essere usata per delimitare qualcosa che va inteso letteralmente e non va confuso con la simbologia usata per la notazione BNF.
nome	I nomi di qualcosa potrebbero essere annotati senza le parentesi angolari, se si usa una forma tipografica particolare per evidenziarli (per esempio in corsivo o in nero, rispetto a un testo normale per ciò che va inteso letteralmente).

Segue un esempio molto semplice, dove si vede l'uso delle parentesi quadre, graffe e dei puntini di sospensione, per descrivere un'istruzione condizionale di un certo linguaggio, senza però entrare troppo nel dettaglio:

```
<istruzione_condizionale> ::=
    IF <espressione_logica>
        THEN
            <sequenza_di_istruzioni>
      [ ELSE
            <sequenza_di_istruzioni> ]
    END FI

<sequenza_di_istruzioni> ::=
    { <istruzione> | <commento> | <riga_bianca> }…
```

Segue lo stesso esempio, modificato in modo da evidenziare i nomi di categoria, evitando così l'uso delle parentesi angolari:

istruzione_condizionale `::= IF` ***espressione_logica***
`THEN`
sequenza_di_istruzioni
`[ ELSE`
sequenza_di_istruzioni `]`
`END FI`

sequenza_di_istruzioni `::= {` ***istruzione*** `|` ***commento*** `|` ***riga_bianca*** `}…`

62.2 Pseudocodifica

«

Un tempo la programmazione avveniva attraverso lunghe fasi di studio a tavolino. Prima di iniziare il lavoro di scrittura del programma (su moduli cartacei che venivano trasferiti successivamente nella

macchina) si passava per la realizzazione di un diagramma di flusso, o *flow chart*.

Il diagramma di flusso andava bene fino a quando si utilizzavano linguaggi di programmazione procedurali, come il COBOL. Quando si sono introdotti concetti nuovi che rendevano tale sistema di rappresentazione più complicato del linguaggio stesso, si è preferito schematizzare gli algoritmi attraverso righe di codice vero e proprio o attraverso una pseudocodifica più o meno adatta al concetto che si vuole rappresentare di volta in volta.

Nelle sezioni successive appaiono esempi realizzati attraverso una pseudocodifica. Tali esempi non sono ottimizzati perché si intende puntare sulla chiarezza piuttosto che sull'eventuale velocità di esecuzione. La pseudocodifica si rifà a termini e concetti comuni a molti linguaggi di programmazione recenti. Vale la pena di chiarire solo alcuni dettagli:

- le variabili di scambio di una subroutine (una procedura o una funzione) vengono semplicemente nominate a fianco del nome della procedura, tra parentesi, cosa che corrisponde a una dichiarazione implicita di quelle variabili con un campo di azione locale e con caratteristiche identiche a quelle usate nelle chiamate relative;
- il trasferimento dei parametri di una chiamata alla subroutine avviene per valore, impedendo l'alterazione delle variabili originali;
- per trasferire una variabile per riferimento, in modo che il suo valore venga aggiornato al termine dell'esecuzione di una sub-

routine, occorre aggiungere il simbolo '`@`' di fronte al nome della variabile utilizzata nella chiamata;

- il simbolo '`#`' rappresenta l'inizio di un commento;
- il simbolo '`:=`' rappresenta l'assegnamento;
- il simbolo '`:==:`' rappresenta lo scambio tra due operandi.

La pseudocodifica in questione non gestisce i puntatori e l'uso dell'operatore «@» è solo un modo per affermare che le modifiche apportate alla variabile devono essere mantenute alla conclusione della funzione.

62.3 Problemi elementari di programmazione

«

Nelle sezioni seguenti sono descritti alcuni problemi elementari attraverso cui si insegnano le tecniche di programmazione ai principianti. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

62.3.1 Somma tra due numeri positivi

«

La somma di due numeri positivi può essere espressa attraverso il concetto dell'incremento unitario: $n+m$ equivale a incrementare m, di un'unità, per n volte, oppure incrementare n per m volte. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli:

```
SOMMA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := X
    FOR I := 1; I <= Y; I++
        Z++
    END FOR

    RETURN Z

END SOMMA
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo enumerativo, '**FOR**'. Il ciclo viene ripetuto '**Y**' volte, incrementando la variabile '**Z**' di un'unità. Alla fine, '**Z**' contiene il risultato della somma di '**X**' per '**Y**'. La pseudocodifica seguente mostra invece la traduzione del ciclo '**FOR**' in un ciclo '**WHILE**':

```
SOMMA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := X
    I := 1
    WHILE I <= Y
        Z++
        I++
    END WHILE

    RETURN Z

END SOMMA
```

62.3.2 Moltiplicazione di due numeri positivi attraverso la somma

«

La moltiplicazione di due numeri positivi, può essere espressa attraverso il concetto della somma: ***n**m*** equivale a sommare ***m*** volte ***n***, oppure ***n*** volte ***m***. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli:

```
MOLTIPLICA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    FOR I := 1; I <= Y; I++
        Z := Z + X
    END FOR

    RETURN Z

END MOLTIPLICA
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo '**FOR**'. Il ciclo viene ripetuto '**Y**' volte, incrementando la variabile '**Z**' del valore di '**X**'. Alla fine, '**Z**' contiene il risultato del prodotto di '**X**' per '**Y**'. La pseudocodifica seguente mostra invece la traduzione del ciclo '**FOR**' in un ciclo '**WHILE**':

```
MOLTIPLICA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    I := 1
    WHILE I <= Y
        Z := Z + X
        I++
    END WHILE

    RETURN Z

END MOLTIPLICA
```

62.3.3 Divisione intera tra due numeri positivi

La divisione di due numeri positivi, può essere espressa attraverso la sottrazione: $n:m$ equivale a sottrarre m da n fino a quando n diventa inferiore di m. Il numero di volte in cui tale sottrazione ha luogo, è il risultato della divisione.

```
DIVIDI (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    I := X
    WHILE I >= Y
        I := I - Y
        Z++
    END WHILE

    RETURN Z

END DIVIDI
```

62.3.4 Elevamento a potenza

«

L'elevamento a potenza, utilizzando numeri positivi, può essere espresso attraverso il concetto della moltiplicazione: ***n*****m*** equivale a moltiplicare ***m*** volte ***n*** per se stesso.

```
EXP (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 1
    FOR I := 1; I <= Y; I++
        Z := Z * X
    END FOR

    RETURN Z

END EXP
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo '**FOR**'. Il ciclo viene ripetuto '**Y**' volte; ogni volta la variabile '**Z**' viene moltiplicata per il valore di '**X**', a partire da uno. Alla fine, '**Z**' contiene il risultato dell'elevamento di '**X**' a '**Y**'. La pseudocodifica seguente mostra invece la traduzione del ciclo '**FOR**' in un ciclo '**WHILE**':

```
EXP (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 1
    I := 1
    WHILE I <= Y
        Z := Z * X
        I++
    END WHILE

    RETURN Z

END EXP
```

La pseudocodifica seguente mostra una soluzione ricorsiva:

```
EXP (X, Y)

    IF X = 0
        THEN
            RETURN 0
        ELSE
            IF Y = 0
                THEN
                    RETURN 1
                ELSE
                    RETURN X * EXP (X, Y-1)
            END IF
    END IF

END EXP
```

62.3.5 Radice quadrata

«

Il calcolo della parte intera della radice quadrata di un numero si può fare per tentativi, partendo da 1, eseguendo il quadrato fino a quando il risultato è minore o uguale al valore di partenza di cui si calcola la radice.

```
RADICE (X)

    LOCAL Z INTEGER
    LOCAL T INTEGER

    Z := 0
    T := 0

    WHILE TRUE

        T := Z * Z
```

```
        IF T > X
            THEN
                # È stato superato il valore massimo.
                Z--
                RETURN Z
        END IF

        Z++

    END WHILE

END RADICE
```

62.3.6 Fattoriale

Il fattoriale è un valore che si calcola a partire da un numero positivo. Può essere espresso come il prodotto di n per il fattoriale di $n-1$, quando n è maggiore di 1, mentre equivale a 1 quando n è uguale a 1. In pratica, $n! = n * (n-1) * (n-2)... * 1$.

```
FATTORIALE (X)

    LOCAL I INTEGER

    I := X - 1

    WHILE I > 0
        X := X * I
        I--
    END WHILE

    RETURN X

END FATTORIALE
```

La soluzione appena mostrata fa uso di un ciclo '**WHILE**' in cui l'indice '**I**', che inizialmente contiene il valore di '**X-1**', viene usato per essere moltiplicato al valore di '**X**', riducendolo ogni volta di un'unità. Quando '**I**' raggiunge lo zero, il ciclo termina e '**X**' contiene il valore del fattoriale. L'esempio seguente mostra invece una soluzione ricorsiva che dovrebbe risultare più intuitiva:

```
FATTORIALE (X)

    IF X == 1
        THEN
            RETURN 1
        ELSE
            RETURN X * FATTORIALE (X - 1)
    END IF

END FATTORIALE
```

62.3.7 Massimo comune divisore

«

Il massimo comune divisore tra due numeri può essere ottenuto sottraendo a quello maggiore il valore di quello minore, fino a quando i due valori sono uguali. Quel valore è il massimo comune divisore.

```
MCD (X, Y)

    WHILE X != Y

        IF X > Y
            THEN
                X := X - Y
            ELSE
                Y := Y - X
        END IF

    END WHILE

    RETURN X

END MCD
```

62.3.8 Numero primo

Un numero intero è numero primo quando non può essere diviso per un altro intero diverso dal numero stesso e da 1, generando un risultato intero.

```
PRIMO (X)

    LOCAL PRIMO BOOLEAN
    LOCAL I INTEGER
    LOCAL J INTEGER

    PRIMO := TRUE
    I := 2

    WHILE (I < X) AND PRIMO

        J := X / I
        J := X - (J * I)
```

```
        IF J == 0
            THEN
                PRIMO := FALSE
            ELSE
                I++
        END IF

    END WHILE

    RETURN PRIMO

END PRIMO
```

62.3.9 Successione di Fibonacci

«

La successione di Fibonacci è una sequenza di numeri interi positivi che hanno la proprietà di essere costituiti dalla somma dei due numeri precedenti nella sequenza stessa. Pertanto, l'n-esimo elemento di questa successione, indicato solitamente come F_n, è dato dalla somma di F_{n-1} e F_{n-2}.

La successione di Fibonacci parte storicamente dal presupposto che F_1 e F_2 siano entrambi pari a uno, ma attualmente si indica anche F_0 pari a zero, cosa che consente di calcolare correttamente F_2.

Per il calcolo della successione di Fibonacci, dall'elemento zero, fino all'elemento n-esimo, vengono proposte due modalità di calcolo, la prima in forma ricorsiva, la seconda in forma iterativa.

```
FIBONACCI (N)
    IF N == 0
    THEN
        RETURN 0
    ELSE
        IF N == 1
        THEN
            RETURN 1
        ELSE
            RETURN (FIBONACCI (N - 1) + FIBONACCI (N - 2))
        END IF
    END IF
END FIBONACCI
```

```
FIBONACCI (N)
    LOCAL F1 := 1
    LOCAL F0 := 0
    LOCAL FN := N
    LOCAL I

    FOR I := 2; I <= N; I++
        FN := F1 + F0
        F0 := F1
        F1 := FN
    END FOR

    RETURN FN

END FIBONACCI
```

La successione di Fibonacci, per cui F_0 è pari a zero e F_1 è pari a uno, è: 0, 1, 1, 2, 3, 5, 8, 13,...

62.4 Scansione di array

«

Nelle sezioni seguenti sono descritti alcuni problemi legati alla scansione di array. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

62.4.1 Ricerca sequenziale

«

La ricerca di un elemento all'interno di un array disordinato può avvenire solo in modo sequenziale, cioè controllando uno per uno tutti gli elementi, fino a quando si trova la corrispondenza cercata. Segue la descrizione delle variabili più importanti che appaiono nella pseudocodifica successiva:

Variabile	Descrizione
`LISTA`	È l'array su cui effettuare la ricerca.
`X`	È il valore cercato all'interno dell'array.
`A`	È l'indice inferiore dell'intervallo di array su cui si vuole effettuare la ricerca.
`Z`	È l'indice superiore dell'intervallo di array su cui si vuole effettuare la ricerca.

Ecco un esempio di pseudocodifica che risolve il problema in modo iterativo:

```
RICERCASEQ (LISTA, X, A, Z)

    LOCAL I INTEGER

    FOR I := A; I <= Z; I++
        IF X == LISTA[I]
            THEN
                RETURN I
        END IF
    END FOR

    # La corrispondenza non è stata trovata.
    RETURN -1

END RICERCASEQ
```

Solo a scopo didattico, viene proposta una soluzione ricorsiva:

```
RICERCASEQ (LISTA, X, A, Z)

    IF A > Z
        THEN
            RETURN -1
        ELSE
            IF X == LISTA[A]
                THEN
                    RETURN A
                ELSE
                    RETURN RICERCASEQ (@LISTA, X, A+1, Z)
            END IF
    END IF

END RICERCASEQ
```

62.4.2 Ricerca binaria

«

La ricerca di un elemento all'interno di un array ordinato può avvenire individuando un elemento centrale: se questo corrisponde all'elemento cercato, la ricerca è terminata, altrimenti si ripete nella parte di array precedente o successiva all'elemento, a seconda del suo valore e del tipo di ordinamento esistente.

Il problema posto in questi termini è ricorsivo. La pseudocodifica mostrata utilizza le stesse variabili già descritte per la ricerca sequenziale.

```
RICERCABIN (LISTA, X, A, Z)

  LOCAL M INTEGER

  # Determina l'elemento centrale dell'array.
  M := (A + Z) / 2

  IF M < A
    THEN
      # Non restano elementi da controllare:
      # l'elemento cercato non c'è.
      RETURN -1
    ELSE
      IF X < LISTA[M]
        THEN
          # Si ripete la ricerca nella parte
          # inferiore.
          RETURN RICERCABIN (@LISTA, X, A, M-1)
        ELSE
          IF X > LISTA[M]
            THEN
              # Si ripete la ricerca nella
              # parte superiore.
```

```
                RETURN RICERCABIN (@LISTA, X, M+1, Z)
              ELSE
                # M rappresenta l'indice
                # dell'elemento cercato.
                RETURN M
            END IF
        END IF
    END IF

END RICERCABIN
```

62.5 Problemi classici di programmazione

Nelle sezioni seguenti sono descritti alcuni problemi classici attraverso cui si insegnano le tecniche di programmazione. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

62.5.1 Bubblesort

Il Bubblesort è un algoritmo relativamente semplice per l'ordinamento di un array, in cui ogni scansione trova il valore giusto per l'elemento iniziale dell'array stesso. Una volta trovata la collocazione di un elemento, si ripete la scansione per il segmento rimanente di array, in modo da collocare un altro valore. La pseudocodifica dovrebbe chiarire il meccanismo.

Variabile	Descrizione
`LISTA`	È l'array da ordinare.
`A`	È l'indice inferiore del segmento di array da ordinare.
`Z`	È l'indice superiore del segmento di array da ordinare.

Viene mostrata una soluzione iterativa:

```
BSORT (LISTA, A, Z)

    LOCAL J INTEGER
    LOCAL K INTEGER

    # Scandisce l'array attraverso l'indice J in modo da
    # collocare ogni volta il valore corretto all'inizio
    # dell'array stesso.
    FOR J := A; J < Z; J++

        # Scandisce l'array attraverso l'indice K scambiando
        # i valori quando sono inferiori a quello di
        # riferimento.
        FOR K := J+1; K <= Z; K++

            IF LISTA[K] < LISTA[J]
                THEN
                    # I valori vengono scambiati.
                    LISTA[K] :==: LISTA[J]
            END IF

        END FOR

    END FOR

END BSORT
```

Segue una soluzione ricorsiva:

```
BSORT (LISTA, A, Z)

    LOCAL K INTEGER

    # L'elaborazione termina quando l'indice inferiore è
```

```
    # maggiore o uguale a quello superiore.
    IF A < Z
        THEN

            # Scandisce l'array attraverso l'indice K
            # scambiando i valori quando sono inferiori a
            # quello iniziale.
            FOR K := A+1; K <= Z; K++

                IF LISTA[K] < LISTA[A]
                    THEN
                        # I valori vengono scambiati.
                        LISTA[K] :==: LISTA[A]
                END IF

            END FOR

            # L'elemento LISTA[A] è collocato correttamente,
            # adesso si ripete la chiamata della funzione in
            # modo da riordinare la parte restante
            # dell'array.
            BSORT (@LISTA, A+1, Z)
    END IF

END BSORT
```

62.5.2 Fusione tra due array ordinati

Due array a una dimensione, con la stessa struttura, ordinati secondo qualche criterio, possono essere fusi in un array singolo, che mantiene l'ordinamento.

Variabile	Descrizione
`A`	È il primo array.
`B`	È il secondo array.
`C`	È l'array da generare con la fusione di '**A**' e '**B**'.
`I`	È l'indice usato per scandire '**A**'.
`J`	È l'indice usato per scandire '**B**'.
`K`	È l'indice usato per scandire '**C**'.
`N`	È la dimensione di '**A**' (l'indice dell'ultimo elemento dell'array).
`M`	È la dimensione di '**B**' (l'indice dell'ultimo elemento dell'array).

Viene mostrata una soluzione iterativa, dove si presume che gli array siano ordinati in modo non decrescente:

```
MERGE (A, N, B, M, C)

    LOCAL I INTEGER
    LOCAL J INTEGER
    LOCAL K INTEGER

    # Si presume che l'indice del primo elemento degli
    # array sia pari a uno.

    I := 1
    J := 1
    K := 1
```

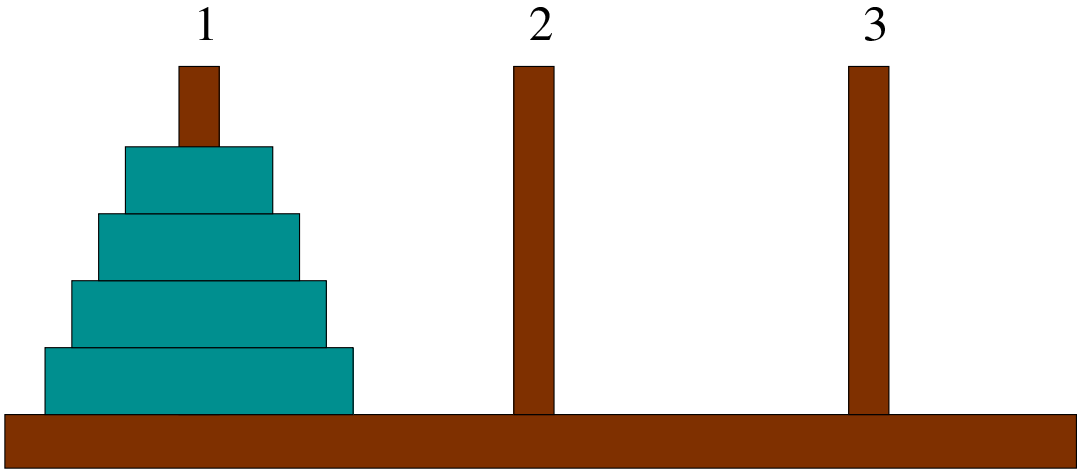
```
    UNTIL I >= N AND J >= M

        IF A(I) <= B(J)
            THEN
                C(K) := A(I)
                I++
            ELSE
                C(K) := B(J)
                J++
        END IF
        K++

    END UNTIL

END MERGE
```

62.5.3 Torre di Hanoi

La torre di Hanoi è un gioco antico: si compone di tre pioli identici conficcati verticalmente su una tavola e di una serie di anelli di larghezze differenti. Gli anelli sono più precisamente dei dischi con un foro centrale che permette loro di essere infilati nei pioli.

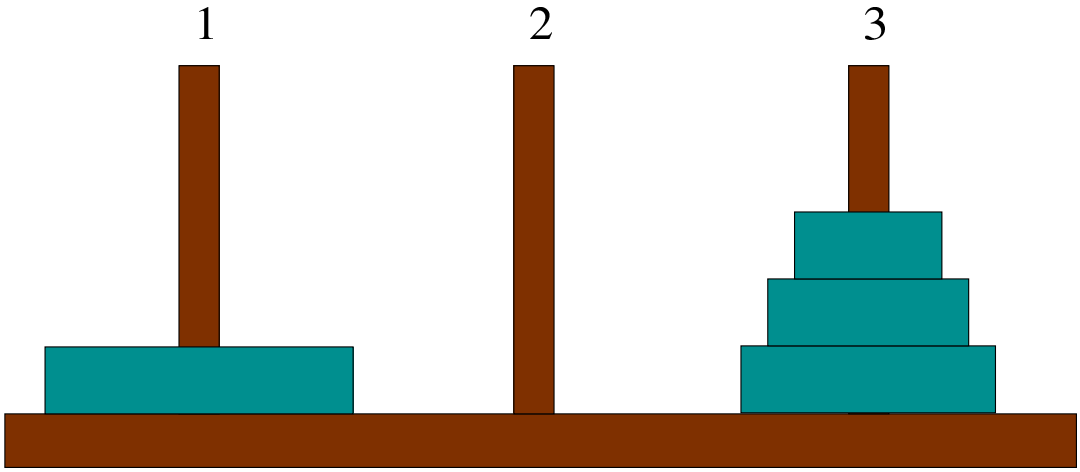
Il gioco inizia con tutti gli anelli collocati in un solo piolo, in ordine, in modo che in basso ci sia l'anello più largo e in alto quello più stretto. Si deve riuscire a spostare tutta la pila di anelli in un dato piolo muovendo un anello alla volta e senza mai collocare un anello più grande sopra uno più piccolo.

Figura 62.27. Situazione iniziale della torre di Hanoi all'inizio del gioco.



Nella figura 62.27 gli anelli appaiono inseriti sul piolo 1; si supponga che questi debbano essere spostati sul piolo 2. Si può immaginare che tutti gli anelli, meno l'ultimo, possano essere spostati in qualche modo corretto, dal piolo 1 al piolo 3, come nella situazione della figura 62.28.

Figura 62.28. Situazione dopo avere spostato $n-1$ anelli.



A questo punto si può spostare l'ultimo anello rimasto (l'n-esimo), dal piolo 1 al piolo 2; quindi, come prima, si può spostare in qualche modo il gruppo di anelli posizionati attualmente nel piolo 3, in modo che finiscano nel piolo 2 sopra l'anello più grande.

Pensando in questo modo, l'algoritmo risolutivo del problema deve essere ricorsivo e potrebbe essere gestito da un'unica subroutine che può essere chiamata opportunamente '**HANOI**'.

Variabile	Descrizione
`N`	È la dimensione della torre espressa in numero di anelli: gli anelli sono numerati da 1 a '**N**'.
`P1`	È il numero del piolo su cui si trova inizialmente la pila di '**N**' anelli.
`P2`	È il numero del piolo su cui deve essere spostata la pila di anelli.
`6-P1-P2`	È il numero dell'altro piolo. Funziona così se i pioli sono numerati da 1 a 3.

Segue la pseudocodifica ricorsiva per la soluzione del problema:

```
HANOI (N, P1, P2)

  IF N > 0
    THEN
      HANOI (N-1, P1, 6-P1-P2)
      scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
      HANOI (N-1, 6-P1-P2, P2)
  END IF
END HANOI
```

Se '**N**', il numero degli anelli da spostare, è minore di 1, non si deve compiere alcuna azione. Se '**N**' è uguale a 1, le istruzioni che dipendono dalla struttura IF-END IF vengono eseguite, ma nessuna delle chiamate ricorsive fa alcunché, dato che '**N-1**' è pari a zero. In questo caso, supponendo che '**N**' sia uguale a 1, che '**P1**' sia pari a 1 e '**P2**' pari a 2, il risultato è semplicemente:

```
Muovi l'anello 1 dal piolo 1 al piolo 2
```

Il risultato è quindi corretto per una pila iniziale consistente di un solo anello.

Se '**N**' è uguale a 2, la prima chiamata ricorsiva sposta un anello ('**N-1**' = 1) dal piolo 1 al piolo 3 (ancora assumendo che i due anelli debbano essere spostati dal primo al terzo piolo) e si sa che questa è la mossa corretta. Quindi viene stampato il messaggio che dichiara lo spostamento del secondo piolo (l'**N**'-esimo) dalla posizione 1 alla posizione 2. Infine, la seconda chiamata ricorsiva si occupa di spostare l'anello collocato precedentemente nel terzo piolo, nel secondo, sopra a quello che si trova già nella posizione finale corretta.

In pratica, nel caso di due anelli che devono essere spostati dal primo al secondo piolo, appaiono i tre messaggi seguenti.

```
Muovi l'anello 1 dal piolo 1 al piolo 3
Muovi l'anello 2 dal piolo 1 al piolo 2
Muovi l'anello 1 dal piolo 3 al piolo 2
```

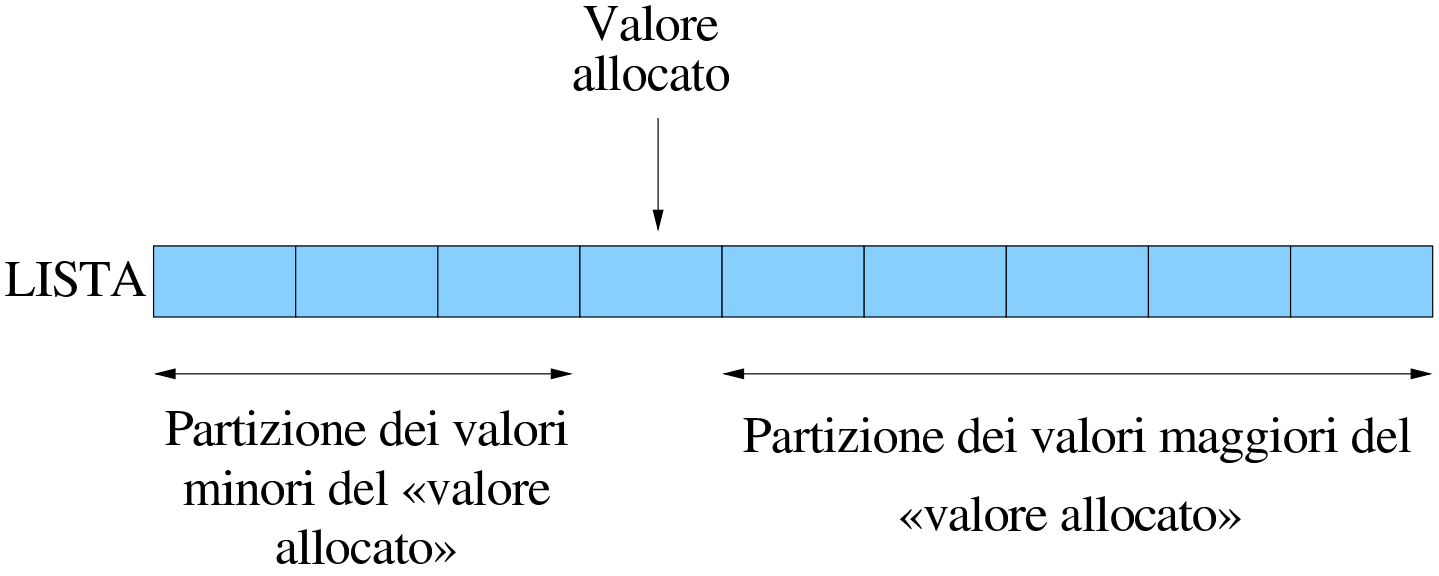
Nello stesso modo si potrebbe dimostrare il funzionamento per un numero maggiore di anelli.

62.5.4 Quicksort (ordinamento non decrescente)

«

L'ordinamento degli elementi di un array è un problema tipico che si può risolvere in tanti modi. Il Quicksort è un algoritmo sofisticato, ottimo per lo studio della gestione degli array, oltre che per quello della ricorsione. Il concetto fondamentale di questo tipo di algoritmo è rappresentato dalla figura 62.33.

Figura 62.33. Il concetto base dell'algoritmo del Quicksort: suddivisione dell'array in due gruppi disordinati, separati da un valore piazzato correttamente nel suo posto rispetto all'ordinamento.



Una sola scansione dell'array è sufficiente per collocare definitivamente un elemento (per esempio il primo) nella sua destinazione finale e allo stesso tempo per lasciare tutti gli elementi con un valore inferiore a quello da una parte, anche se disordinati, e tutti quelli con un valore maggiore, dall'altra.

In questo modo, attraverso delle chiamate ricorsive, è possibile elaborare i due segmenti dell'array rimasti da riordinare.

L'algoritmo può essere descritto grossolanamente come:

1. localizzazione della collocazione finale del primo valore, separando in questo modo i valori;
2. ordinamento del segmento precedente all'elemento collocato definitivamente;
3. ordinamento del segmento successivo all'elemento collocato

definitivamente.

Viene qui indicato con '**PART**' la subroutine che esegue la scansione dell'array, o di un suo segmento, per determinare la collocazione finale (indice '**CF**') del primo elemento (dell'array o del segmento in questione).

Sia '**LISTA**' l'array da ordinare. Il primo elemento da collocare corrisponde inizialmente a '**LISTA[A]**' e il segmento di array su cui intervenire corrisponde a '**LISTA[A:Z]**' (cioè a tutti gli elementi che vanno dall'indice '**A**' all'indice '**Z**').

Alla fine della prima scansione, l'indice '**CF**' rappresenta la posizione in cui occorre spostare il primo elemento, cioè '**LISTA[A]**'. In pratica, '**LISTA[A]**' e '**LISTA[CF]**' vengono scambiati.

Durante la scansione che serve a determinare la collocazione finale del primo elemento, '**PART**' deve occuparsi di spostare gli elementi prima o dopo quella posizione, in funzione del loro valore, in modo che alla fine quelli inferiori o uguali a quello dell'elemento da collocare si trovino nella parte inferiore e gli altri dall'altra. In pratica, alla fine della prima scansione, gli elementi contenuti in '**LISTA[A:(CF-1)]**' devono contenere valori inferiori o uguali a '**LISTA[CF]**', mentre quelli contenuti in '**LISTA[(CF+1):Z]**' devono contenere valori superiori.

Indichiamo con '**QSORT**' la subroutine che esegue il compito complessivo di ordinare l'array. Il suo lavoro consisterebbe nel chiamare '**PART**' per collocare il primo elemento, continuando poi con la chiamata ricorsiva di se stessa per la parte di array precedente all'elemento collocato e infine alla chiamata ricorsiva per la parte restante di array.

Assumendo che '**PART**' e le chiamate ricorsive di '**QSORT**' svolgano il loro compito correttamente, si potrebbe fare un'analisi informale dicendo che se l'indice '**Z**' **non** è maggiore di '**A**', allora c'è un elemento (o nessuno) all'interno di '**LISTA[A:Z]**' e inoltre, '**LISTA[A:Z]**' è già nel suo stato finale. Se '**Z**' è maggiore di '**A**', allora (per assunzione) '**PART**' ripartisce correttamente '**LISTA[A:Z]**'. L'ordinamento separato dei due segmenti (per assunzione eseguito correttamente dalle chiamate ricorsive) completa l'ordinamento di '**LISTA[A:Z]**'.

Le figure 62.34 e 62.35 mostrano due fasi della scansione effettuata da '**PART**' all'interno dell'array o del segmento che gli viene fornito.

Figura 62.34. La scansione dell'array da parte di '**PART**' avviene portando in avanti l'indice '**I**' e portando indietro l'indice '**CF**'. Quando l'indice '**I**' localizza un elemento che contiene un valore maggiore di '**LISTA[A]**' e l'indice '**CF**' localizza un elemento che contiene un valore inferiore o uguale a '**LISTA[A]**', gli elementi cui questi indici fanno riferimento vengono scambiati, quindi il processo di avvicinamento tra '**I**' e '**CF**' continua.

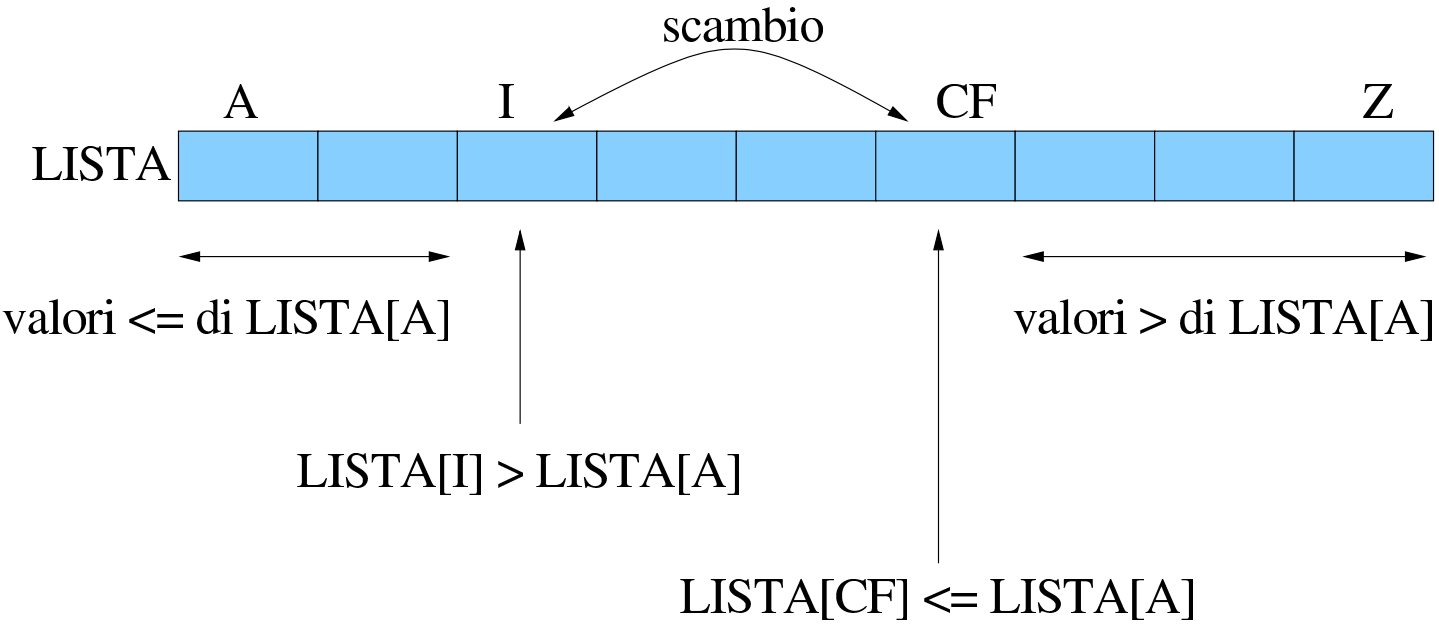
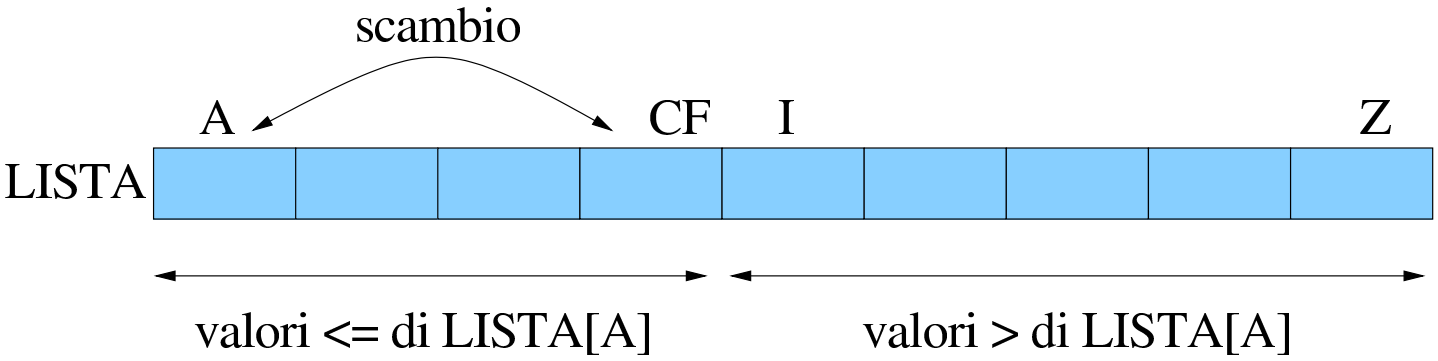


Figura 62.35. Quando la scansione è giunta al termine, quello che resta da fare è scambiare l'elemento '**LISTA[A]**' con '**LISTA[CF]**'.



In pratica, l'indice '**I**', iniziando dal valore '**A+1**', viene spostato verso destra fino a che viene trovato un elemento maggiore di '**LISTA[A]**', quindi è l'indice '**CF**' a essere spostato verso sinistra, iniziando dalla stessa posizione di '**Z**', fino a che viene incontrato un elemento minore o uguale a '**LISTA[A]**'. Questi elementi vengono scambiati e lo spostamento di '**I**' e '**CF**' riprende. Ciò prosegue fino a che '**I**' e '**CF**' si incontrano, momento in cui '**LISTA[A:Z]**' è stata ripartita e '**CF**' rappresenta l'indice di un elemento che si trova nella sua collocazione finale.

Variabile	Descrizione
`LISTA`	L'array da ordinare in modo crescente.
`A`	L'indice inferiore del segmento di array da ordinare.
`Z`	L'indice superiore del segmento di array da ordinare.
`CF`	Sta per «collocazione finale» ed è l'indice che cerca e trova la posizione giusta di un elemento nell'array.
`I`	È l'indice che insieme a '**CF**' serve a ripartire l'array.

Segue la pseudocodifica delle due subroutine:

```
PART (LISTA, A, Z)

    LOCAL I INTEGER
    LOCAL CF INTEGER

    # si assume che A < Z

    I := A + 1
    CF := Z

    WHILE TRUE # ciclo senza fine.

        WHILE TRUE

            # sposta I a destra

            IF ((LISTA[I] > LISTA[A]) OR (I >= CF))
                THEN
                    BREAK
                ELSE
                    I := I + 1
            END IF

        END WHILE

        WHILE TRUE

            # sposta CF a sinistra

            IF (LISTA[CF] <= LISTA[A])
                THEN
                    BREAK
                ELSE
                    CF := CF - 1
```

```
            END IF

        END WHILE

        IF CF <= I
            THEN
                # è avvenuto l'incontro tra I e CF
                BREAK
            ELSE
                # vengono scambiati i valori
                LISTA[CF] :==: LISTA[I]
                I := I + 1
                CF := CF - 1
        END IF

    END WHILE

    # a questo punto LISTA[A:Z] è stata ripartita e CF è la
    # collocazione di LISTA[A]

    LISTA[CF] :==: LISTA[A]

    # a questo punto, LISTA[CF] è un elemento (un valore)
    # nella giusta posizione

    RETURN CF

END PART
```

```
QSORT (LISTA, A, Z)

    LOCAL CF INTEGER

    IF Z > A
        THEN
            CF := PART (@LISTA, A, Z)
            QSORT (@LISTA, A, CF-1)
            QSORT (@LISTA, CF+1, Z)
    END IF
END QSORT
```

Vale la pena di osservare che l'array viene indicato nelle chiamate in modo che alla subroutine sia inviato un riferimento a quello originale, perché le variazioni fatte all'interno delle subroutine devono riflettersi sull'array originale.

62.5.5 Permutazioni

La permutazione è lo scambio di un gruppo di elementi posti in sequenza. Il problema che si vuole analizzare è la ricerca di tutte le permutazioni possibili di un dato gruppo di elementi.

Se ci sono n elementi in un array, allora alcune delle permutazioni si possono ottenere bloccando l'n-esimo elemento e generando tutte le permutazioni dei primi $n-1$ elementi. Quindi l'n-esimo elemento può essere scambiato con uno dei primi $n-1$, ripetendo poi la fase precedente. Questa operazione deve essere ripetuta finché ognuno degli n elementi originali è stato usato nell'n-esima posizione.

Variabile	Descrizione
`LISTA`	L'array da permutare.
`A`	L'indice inferiore del segmento di array da permutare.
`Z`	L'indice superiore del segmento di array da permutare.
`K`	È l'indice che serve a scambiare gli elementi.

Segue la pseudocodifica:

```
PERMUTA (LISTA, A, Z)

    LOCAL K INTEGER
    LOCAL N INTEGER

    IF (Z - A) >= 1
        # Ci sono almeno due elementi nel segmento di array.
        THEN
            FOR K := Z; K >= A; K--

                LISTA[K] :==: LISTA[Z]

                PERMUTA (LISTA, A, Z-1)

                LISTA[K] :==: LISTA[Z]

            END FOR
        ELSE
            scrivi LISTA
    END IF

END PERMUTA
```

62.6 Gestione dei file

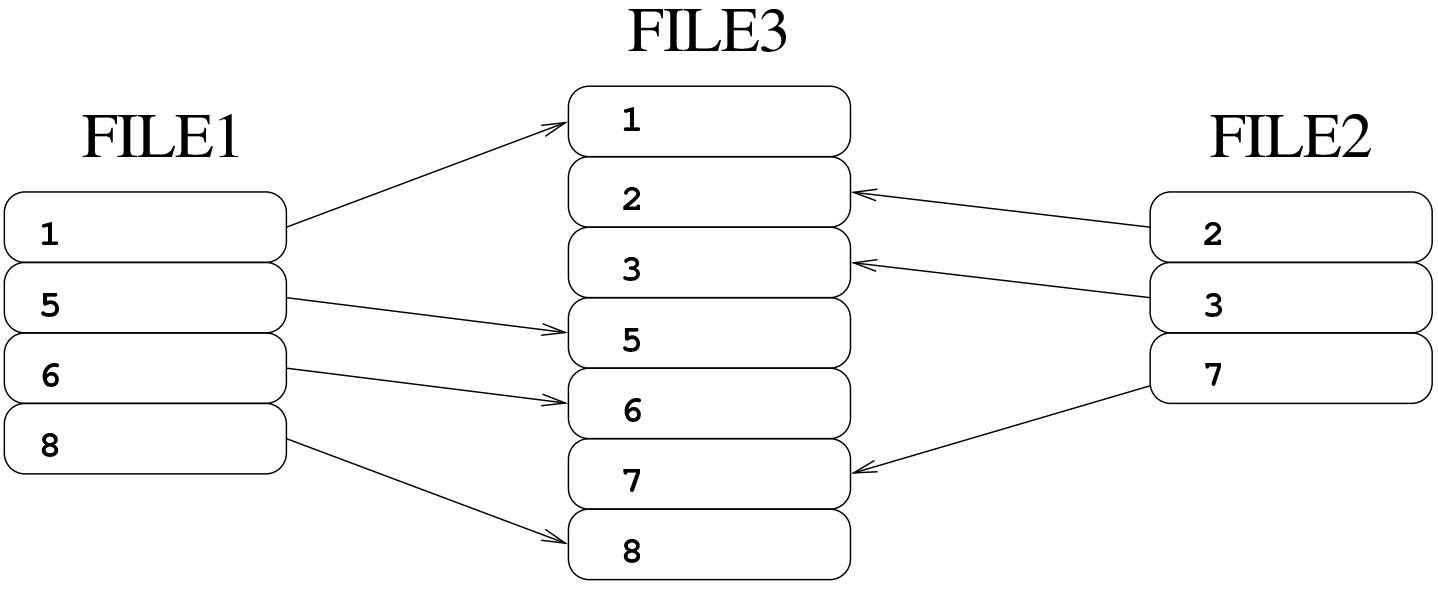
La gestione dei file è sempre la questione più complessa nello studio degli algoritmi, quanto si esclude la possibilità di gestire semplicemente tutto nella memoria centrale. In questo tipo di situazione, il file deve essere inteso come un insieme di record, che spesso sono di dimensione uniforme.

Nelle sezioni seguenti sono descritti alcuni problemi legati alla gestione dei file, con la soluzione in forma di pseudocodifica.

62.6.1 Fusione tra due file ordinati

La fusione di due file ordinati, aventi la stessa struttura, avviene leggendo un record da entrambi i file, confrontando la chiave di ordinamento e scrivendo nel file da ottenere il record con chiave più bassa. Successivamente, viene letto solo record successivo del file che conteneva la chiave più bassa e si ripete il confronto.

Figura 62.41. Il file '`FILE1`' e '`FILE2`' vengono fusi per generare il file '`FILE3`'.



Segue un esempio di pseudocodifica per la soluzione del problema della fusione tra due file. La funzione riceve il riferimento ai file da elaborare e si può osservare l'utilizzo di variabili strutturate per accogliere i record dei file da elaborare. Come si può intuire, le variabili booleane il cui nome inizia per '**EOF**', rappresentano l'avverarsi della condizione di «fine del file»; in pratica, quando contengono il valore *Vero*, indicano che la lettura del file a cui si riferiscono è andata oltre la conclusione del file.

```
FUSIONE_DUE_FILE (FILE_IN_1, FILE_IN_2, FILE_OUT)

  LOCAL RECORD_1:
        CHIAVE_1       CHARACTER (8)
        DATI_1         CHARACTER (72)
  LOCAL EOF_1 := FALSE

  LOCAL RECORD_2:
        CHIAVE_2       CHARACTER (8)
        DATI_2         CHARACTER (72)
  LOCAL EOF_2 := FALSE

  LOCAL RECORD_3       CHARACTER (80)

  OPEN INPUT  FILE_IN_1
  OPEN INPUT  FILE_IN_2
  OPEN OUTPUT FILE_OUT

  READ FILE_IN_1 NEXT RECORD INTO RECORD_1
  IF END OF FILE
    THEN
      EOF_1 := TRUE
  END IF

  READ FILE_IN_2 NEXT RECORD INTO RECORD_2
```

```
IF END OF FILE
  THEN
    EOF_2 := TRUE
END IF

UNTIL EOF_1 AND EOF_2

  IF EOF_1
    THEN
      RECORD_3 := RECORD_2
      READ FILE_IN_2 NEXT RECORD INTO RECORD_2
      IF END OF FILE
        THEN
          EOF_2 := TRUE
      END IF
    ELSE
      IF EOF_2
        THEN
          RECORD_3 := RECORD_1
          READ FILE_IN_1 NEXT RECORD INTO RECORD_1
          IF END OF FILE
            THEN
              EOF_1 := TRUE
          END IF
        ELSE
          IF CHIAVE_1 < CHIAVE_2
            THEN
              RECORD_3 := RECORD_1
              READ FILE_IN_1 NEXT RECORD INTO RECORD_1
              IF END OF FILE
                THEN
                  EOF_1 := TRUE
              END IF
            ELSE
```

```
                    RECORD_3 := RECORD_2
                    READ FILE_IN_2 NEXT RECORD INTO RECORD_2
                    IF END OF FILE
                      THEN
                        EOF_2 := TRUE
                    END IF
                  END IF
            END IF
      END IF

      WRITE FILE_OUT RECORD FROM RECORD_3

   END UNTIL

   CLOSE FILE_IN_1
   CLOSE FILE_IN_2
   CLOSE FILE_OUT

END FUSIONE_DUE_FILE
```

62.6.2 Riordino attraverso la fusione

«

Un file non ordinato può essere ordinato, attraverso una serie di passaggi, che prevedono la divisione in due parti del file (ovvero ***biforcazione***), contenenti le raccolte dei blocchi di record che risultano essere nella sequenza corretta, per poi fondere queste due parti e ripetere il procedimento. Le figure successive mostrano le due fasi: la separazione in due file, la fusione dei due file. Se il file risultante non è ordinato completamente, occorre procedere con una nuova fase di separazione e fusione. Si osservi, in particolare, che nelle figure, il file iniziale contiene solo tre blocchi di record in sequenza, pertanto,

l'ultimo di questi blocchi viene collocato nel file finale senza una fusione con un blocco corrispondente.

Figura 62.43. Biforcazione del file.

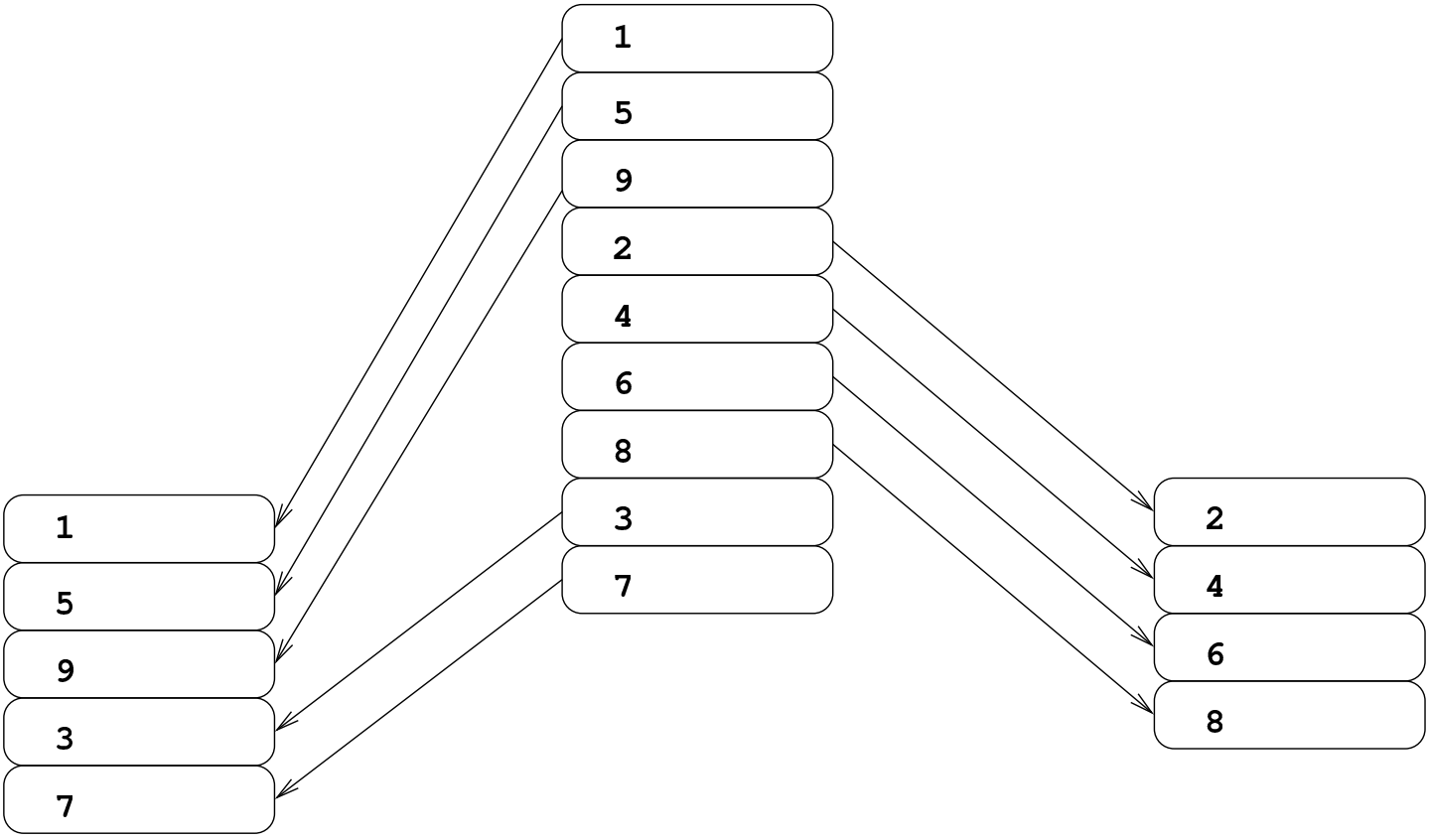
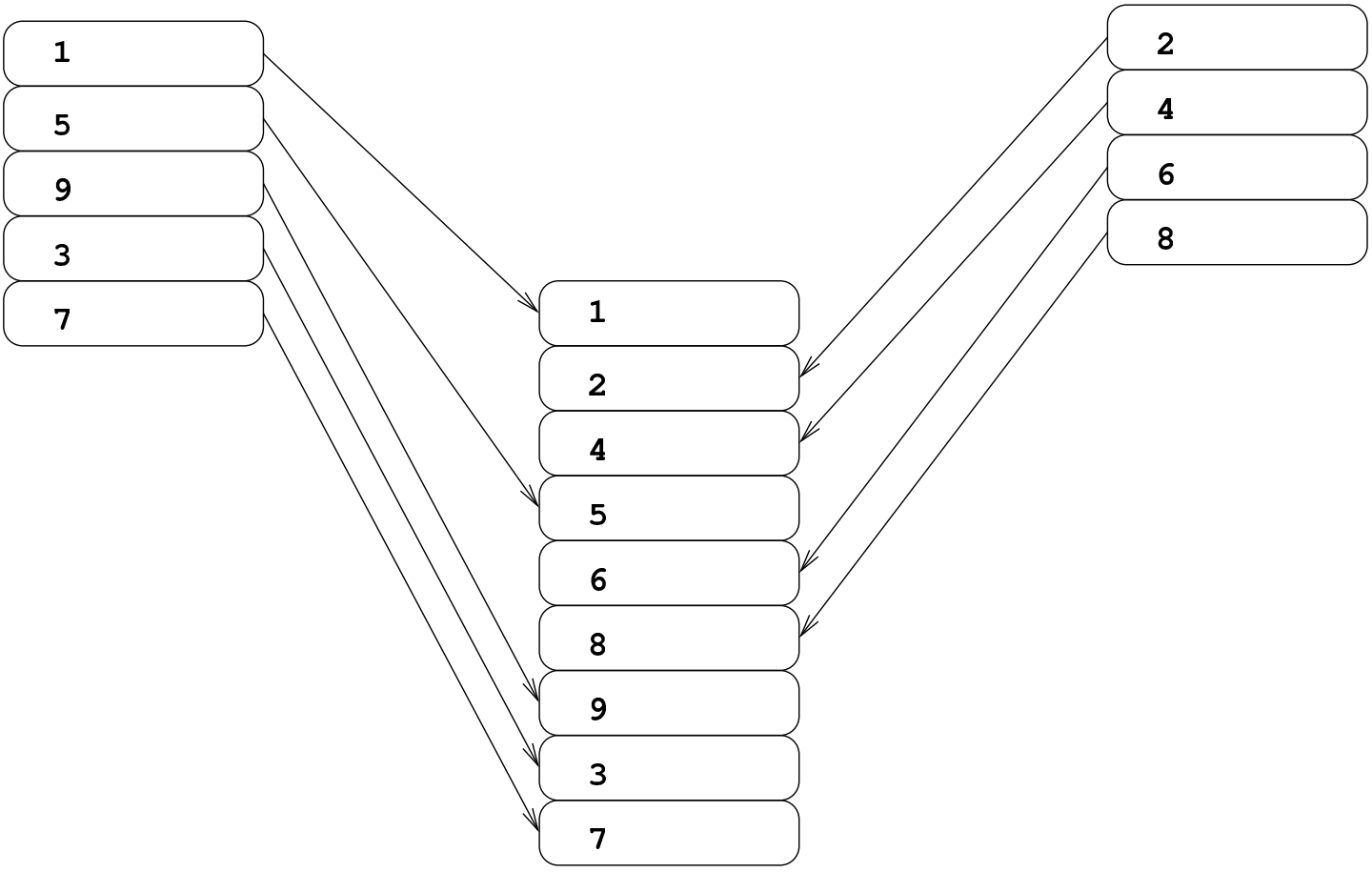


Figura 62.44. Fusione.



Per costruire un programma che utilizza questa tecnica di ordinamento, si può considerare che sia avvenuta l'ultima fase di biforcazione quando si contano un massimo di due blocchi; pertanto, la fase successiva di fusione produce sicuramente il file ordinato finale.

Segue un esempio di pseudocodifica per eseguire la biforcazione. Si osservi che i nomi dei file vengono passati in qualche modo, così che dopo la chiamata di questa procedura sia possibile riaprire tali file per la fusione; inoltre, l'informazione contenuta nella variabile ***BIFORCAZIONI***, viene restituita come valore della chiamata della funzione, in modo da poter conoscere, dopo la chiamata, quante separazioni sono state eseguite nel file di partenza.

```
BIFORCA (FILE_IN_1, FILE_OUT_1, FILE_OUT_2)
```

```
LOCAL RECORD_IN_1:
      CHIAVE         CHARACTER (8)
      DATI           CHARACTER (72)
LOCAL CHIAVE_ORIG    CHARACTER (8)
LOCAL EOF_1 := FALSE

LOCAL RECORD_OUT_1   CHARACTER (80)
LOCAL RECORD_OUT_2   CHARACTER (80)

LOCAL SCAMBIO := 1
LOCAL BIFORCAZIONI := 0

OPEN INPUT  FILE_IN_1
OPEN OUTPUT FILE_OUT_1
OPEN OUTPUT FILE_OUT_2

READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
IF END OF FILE
  THEN
    EOF_1 := TRUE
  ELSE
    BIFORCAZIONI++
    WRITE FILE_OUT_1 RECORD FROM RECORD_IN_1
    CHIAVE_ORIG := CHIAVE
    READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
    IF END OF FILE
      THEN
        EOF_1 := TRUE
    END IF
END IF

UNTIL EOF_1
```

```
    IF CHIAVE >= CHIAVE_ORIG
      THEN
        IF SCAMBIO == 1
          THEN
            WRITE FILE_OUT_1 RECORD FROM RECORD_IN_1
            CHIAVE_ORIG := CHIAVE
            READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
            IF END OF FILE
              THEN
                EOF_1 := TRUE
            END IF
          ELSE
            WRITE FILE_OUT_2 RECORD FROM RECORD_IN_1
            CHIAVE_ORIG := CHIAVE
            READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
            IF END OF FILE
              THEN
                EOF_1 := TRUE
            END IF
        END IF
      ELSE
        BIFORCAZIONI++
        CHIAVE_ORIG := CHIAVE
        IF SCAMBIO == 1
          THEN
            SCAMBIO := 2
          ELSE
            SCAMBIO := 1
        END IF
    END IF
  END UNTIL

  CLOSE FILE_IN_1
  CLOSE FILE_OUT_1
```

```
  CLOSE FILE_OUT_2

  RETURN BIFORCAZIONI

END BIFORCA
```

Segue un esempio di pseudocodifica per eseguire la fusione a blocchi. Si osservi che i nomi dei file vengono passati in qualche modo, così che dopo la chiamata di questa procedura sia possibile riaprire tali file per la biforcazione e di nuovo per la fusione. Come si può intuire, le variabili booleane il cui nome inizia per '**EOB**', rappresentano l'avverarsi della condizione di «fine del blocco» non decrescente; in pratica, quando contengono il valore *Vero*, indicano che la lettura del file a cui si riferiscono ha prodotto un record che ha una chiave inferiore rispetto a quello letto precedentemente, oppure che non sono disponibili altri record.

```
FUSIONE (FILE_IN_1, FILE_IN_2, FILE_OUT)

  LOCAL RECORD_1:
        CHIAVE_1       CHARACTER (8)
        DATI_1         CHARACTER (72)
  LOCAL CHIAVE_1_ORIG CHARACTER (8)
  LOCAL EOF_1 := FALSE
  LOCAL EOB_1 := FALSE

  LOCAL RECORD_2:
        CHIAVE_2       CHARACTER (8)
        DATI_2         CHARACTER (72)
  LOCAL CHIAVE_2_ORIG CHARACTER (8)
  LOCAL EOF_2 := FALSE
  LOCAL EOB_2 := FALSE
```

```
LOCAL RECORD_3      CHARACTER (80)

OPEN INPUT  FILE_IN_1
OPEN INPUT  FILE_IN_2
OPEN OUTPUT FILE_OUT

READ FILE_IN_1 NEXT RECORD INTO RECORD_1
IF END OF FILE
  THEN
    EOF_1 := TRUE
    EOB_1 := TRUE
  ELSE
    CHIAVE_1_ORIG := CHIAVE_1
END IF

READ FILE_IN_2 NEXT RECORD INTO RECORD_2
IF END OF FILE
  THEN
    EOF_2 := TRUE
    EOB_2 := TRUE
  ELSE
    CHIAVE_2_ORIG := CHIAVE_2
END IF

UNTIL EOF_1 AND EOF_2
  UNTIL EOB_1 AND EOB_2

    IF EOB_1
      THEN
        RECORD_3 := RECORD_2
        READ FILE_IN_2 NEXT RECORD INTO RECORD_2
        IF END OF FILE
          THEN
            EOF_2 := TRUE
```

```
            EOB_2 := TRUE
          END IF
        ELSE
          IF EOB_2
            THEN
              RECORD_3 := RECORD_1
              READ FILE_IN_1 NEXT RECORD INTO RECORD_1
              IF END OF FILE
                THEN
                  EOF_1 := TRUE
                  EOB_1 := TRUE
              END IF
            ELSE
              IF CHIAVE_1 < CHIAVE_2
                THEN
                  RECORD_3 := RECORD_1
                  READ FILE_IN_1 NEXT RECORD INTO RECORD_1
                  IF END OF FILE
                    THEN
                      EOF_1 := TRUE
                      EOB_1 := TRUE
                    ELSE
                      IF CHIAVE_1 >= CHIAVE_1_ORIG
                        THEN
                          CHIAVE_1_ORIG := CHIAVE_1
                        ELSE
                          EOB_1 := TRUE
                    END IF
                  END IF
                ELSE
                  RECORD_3 := RECORD_2
                  READ FILE_IN_2 NEXT RECORD INTO RECORD_2
                  IF END OF FILE
                    THEN
```

```
                        EOF_2 := TRUE
                        EOB_2 := TRUE
                      ELSE
                        IF CHIAVE_2 >= CHIAVE_2_ORIG
                          THEN
                            CHIAVE_2_ORIG := CHIAVE_2
                          ELSE
                            EOB_2 := TRUE
                        END IF
                    END IF
                END IF
            END IF
        END IF

        WRITE FILE_OUT RECORD FROM RECORD_3

    END UNTIL

    IF NOT EOF_1
    THEN
        EOB_1 := FALSE
    END IF

    IF NOT EOF_2
    THEN
        EOB_2 := FALSE
    END IF

END UNTIL

CLOSE FILE_IN_1
CLOSE FILE_IN_2
CLOSE FILE_OUT
```

```
END FUSIONE
```

Per poter riordinare effettivamente un file, utilizzando le procedure descritte, si può utilizzare la pseudocodifica seguente, che si avvale di quanto già descritto. Per non dover mostrare nella pseudocodifica come si dichiarano i file, si suppone che questo sia compito di un'altra porzione di codice assente, nel quale si chiama la procedura sottostante, indicando i riferimenti ai file da utilizzare. Si osservi che il file originale non viene modificato, producendo eventualmente un altro file ordinato.

```
ORDINAMENTO (FILE_IN, FILE_TMP_1, FILE_TMP_2, FILE_OUT)

  LOCAL BIFORCAZIONI

  BIFORCAZIONI := BIFORCA (@FILE_IN_1, @FILE_TMP_1,
                           @FILE_TMP_2)
  #
  # se la variabile BIFORCAZIONI contiene zero, significa
  # che il file è vuoto.
  #
  IF BIFORCAZIONI > 0
    THEN
      FUSIONE (@FILE_TMP_1, @FILE_TMP_2, @FILE_OUT)
      WHILE BIFORCAZIONI > 2
        BIFORCAZIONI := BIFORCA (@FILE_OUT, @FILE_TMP_1,
                                 @FILE_TMP_2)
        FUSIONE (@FILE_TMP_1, @FILE_TMP_2, @FILE_OUT)
      END WHILE
  END IF

END ORDINAMENTO
```

62.7 Trasformazione in lettere

«

Nell'ambito della realizzazione di applicativi gestionali, capitano frequentemente problemi di conversione di numeri interi in una qualche forma alfabetica. In queste sezioni vengono mostrati degli algoritmi molto semplici per risolvere questo tipo di problemi.

62.7.1 Da numero a sequenza alfabetica pura

«

Esiste un tipo di numerazione in cui si utilizzano solo le lettere dell'alfabeto, dalla «a» alla «z», senza accenti o altri simboli speciali, senza distinguere tra maiuscole e minuscole. In generale, i simboli da «a» a «z» consentono di rappresentare valori da 1 a 26, dove lo zero è escluso. Per rappresentare valori superiori, si possono accoppiare più lettere, ma il calcolo non è banale, proprio perché manca lo zero.

Attraverso la pseudocodifica introdotta nella sezione 62.2, si può descrivere una funzione che calcoli la stringa corrispondente a un numero intero positivo, maggiore di zero. Per prima cosa, occorre definire una sotto funzione che sia in grado di trasformare un numero intero, compreso tra 1 e 26 nella lettera alfabetica corrispondente; ciò si ottiene abbastanza facilmente, attraverso la verifica di più condizioni in cascata. Il vero problema, purtroppo, sta nel costruire una stringa composta da più lettere, quando si vuole rappresentare un valore superiore a 26. Non essendoci lo zero, diventa difficile fare i calcoli. Se si parte dal presupposto che il numero da convertire non possa essere superiore a 702, si sa con certezza che servono al massimo due lettere alfabetiche (perché la stringa «ZZ» corrisponderebbe proprio al numero 702); in tal caso, è sufficiente dividere il numero per 26, dove la parte intera rappresenta la prima lettera, mentre il re-

sto rappresenta la seconda. Tuttavia, se la divisione non dà resto, la stringa corretta è quella precedente. Per esempio, il numero 53 corrisponde alla stringa «BA», perché 53/26 = 2 con un resto di 1. Nello stesso modo, però, 52 si traduce nella stringa «AZ», perché 53/26 = 2, ma non c'è resto, pertanto, «B_» diventa «AZ».

La pseudocodifica seguente riepiloga il concetto in modo semplificato, dove, a seconda delle esigenze, la conversione è sempre limitata a un valore massimo. Le omissioni sono parti di codice facilmente intuibili.

```
INTEGER_TO_ALPHABET (N)

    ALPHABET_DIGIT (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "A"
        ELSE IF DIGIT = 2
            THEN
                RETURN "B"
        …
        …
        ELSE IF DIGIT = 26
            THEN
                RETURN "Z"
        ELSE
            RETURN "##ERROR##"
        FI
    END ALPHABET_DIGIT

    IF N <= 0
```

```
        THEN
            RETURN "##ERROR##";
    ELSE IF N <= 26
        THEN
            RETURN ALPHABET_DIGIT (N)
    ELSE IF N <= 52
        THEN
            N := N - 52
            RETURN "A" ALPHABET_DIGIT (N)
    ELSE IF N <= 78
        THEN
            N := N - 78
            RETURN "B" ALPHABET_DIGIT (N)
    …
    …
    ELSE IF N <= 702
        THEN
            N := N - 702
            RETURN "Z" ALPHABET_DIGIT (N)
    ELSE IF N <= 728
        THEN
            N := N - 728
            RETURN "AA" ALPHABET_DIGIT (N)
    ELSE IF N <= 754
        THEN
            N := N - 754
            RETURN "AB" ALPHABET_DIGIT (N)
    …
    …
    ELSE
        RETURN "##ERROR##"
    END IF
END INTEGER_TO_ALPHABET
```

62.7.2 Da numero a numero romano

La conversione di un numero intero positivo in una stringa che rappresenta un numero romano, ha un discreto livello di difficoltà, perché la numerazione romana non prevede lo zero, perché la tecnica prevede la somma e la sottrazione di simboli (a seconda della posizione) e poi perché diventa difficile indicare valori multipli delle migliaia.

Per prima cosa è necessario conoscere il valore associato ai simboli elementari:

Simbolo	Valore corrispondente
I	1
V	5
X	10
L	50
C	100
D	500
M	1 000

Un simbolo posto alla destra di un altro simbolo con un valore maggiore o uguale di questo, viene sommato; al contrario, un simbolo posto alla sinistra di un altro simbolo con un valore maggiore di questo, viene sottratto. Per esempio, «VI» equivale a 5+1, mentre «IV» equivale a 5−1. Esistono comunque anche altri vincoli, per evitare di creare numeri difficili da interpretare a causa di una complessità di calcolo eccessiva.

Per risolvere il problema con un algoritmo relativamente semplice, si può scomporre il valore di partenza in fasce: unità, decine, centinaia e migliaia (la conversione di valori superiori genererebbe sol-

tanto una serie lunghissima di «M» che risulta poi troppo difficile da leggere).

```
INTEGER_TO_ROMAN (N)

    LOCAL DIGIT_1 INTEGER
    LOCAL DIGIT_2 INTEGER
    LOCAL DIGIT_3 INTEGER
    LOCAL DIGIT_4 INTEGER

    DIGIT_1 := 0
    DIGIT_2 := 0
    DIGIT_3 := 0
    DIGIT_4 := 0

    DIGIT_1_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "I"
        ELSE IF DIGIT = 2
            THEN
                RETURN "II"
        ELSE IF DIGIT = 3
            THEN
                RETURN "III"
        ELSE IF DIGIT = 4
            THEN
                RETURN "IV"
        ELSE IF DIGIT = 5
            THEN
                RETURN "V"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "VI"
        ELSE IF DIGIT = 7
            THEN
                RETURN "VII"
        ELSE IF DIGIT = 8
            THEN
                RETURN "VIII"
        ELSE IF DIGIT = 9
            THEN
                RETURN "IX"
        END IF
    END DIGIT_1_TO_ROMAN

    DIGIT_2_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "X"
        ELSE IF DIGIT = 2
            THEN
                RETURN "XX"
        ELSE IF DIGIT = 3
            THEN
                RETURN "XXX"
        ELSE IF DIGIT = 4
            THEN
                RETURN "XL"
        ELSE IF DIGIT = 5
            THEN
                RETURN "L"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "LX"
        ELSE IF DIGIT = 6
            THEN
                RETURN "LXX"
        ELSE IF DIGIT = 8
            THEN
                RETURN "LXXX"
        ELSE IF DIGIT = 9
            THEN
                RETURN "XC"
        END IF
    END DIGIT_2_TO_ROMAN

    DIGIT_3_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "C"
        ELSE IF DIGIT = 2
            THEN
                RETURN "CC"
        ELSE IF DIGIT = 3
            THEN
                RETURN "CCC"
        ELSE IF DIGIT = 4
            THEN
                RETURN "CD"
        ELSE IF DIGIT = 5
            THEN
                RETURN "D"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "DC"
        ELSE IF DIGIT = 7
            THEN
                RETURN "DCC"
        ELSE IF DIGIT = 8
            THEN
                RETURN "DCCC"
        ELSE IF DIGIT = 9
            THEN
                RETURN "CM"
        END IF
    END DIGIT_3_TO_ROMAN

    DIGIT_4_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "M"
        ELSE IF DIGIT = 2
            THEN
                RETURN "MM"
        ELSE IF DIGIT = 3
            THEN
                RETURN "MMM"
        ELSE IF DIGIT = 4
            THEN
                RETURN "MMMM"
        ELSE IF DIGIT = 5
            THEN
                RETURN "MMMMM"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "MMMMMM"
        ELSE IF DIGIT = 7
            THEN
                RETURN "MMMMMMM"
        ELSE IF DIGIT = 8
            THEN
                RETURN "MMMMMMMM"
        ELSE IF DIGIT = 9
            THEN
                RETURN "MMMMMMMMM"
        END IF
    END DIGIT_4_TO_ROMAN

    DIGIT_4 := int (N/1000)
    N := N - (DIGIT_4 * 1000)

    DIGIT_3 := int (N/100)
    N := N - (DIGIT_3 * 100)

    DIGIT_2 := int (N/10)
    N := N - (DIGIT_2 * 10)

    DIGIT_1 := N

    RETURN DIGIT_4_TO_ROMAN (DIGIT_4)
           DIGIT_3_TO_ROMAN (DIGIT_3)
           DIGIT_2_TO_ROMAN (DIGIT_2)
           DIGIT_1_TO_ROMAN (DIGIT_2)

END INTEGER_TO_ROMAN
```

Come si vede, dopo aver scomposto il valore in quattro fasce, si utilizzano quattro funzioni distinte per ottenere la porzione di stringa

che traduce il valore relativo. L'istruzione '**RETURN**' finale intende concatenare tutte le stringhe risultanti.

62.7.3 Da numero a lettere, nel senso verbale

Quando si trasforma un numero in lettere, per esempio quando si vuole trasformare 123 in «centoventitre», l'algoritmo di conversione deve tenere conto delle convenzioni linguistiche e non esiste una soluzione generale per tutte le lingue.

Per quanto riguarda la lingua italiana, esistono nomi diversi fino al 19, poi ci sono delle particolarità per i plurali o i singolari. La pseudocodifica seguente risolve il problema in una sola funzione ricorsiva. Le omissioni dovrebbero essere sufficientemente intuitive.

```
INTEGER_TO_ITALIAN (N)

  LOCAL X INTEGER
  LOCAL Y INTEGER

  IF N = 0
    THEN
      RETURN ""
  ELSE IF N = 1
    THEN
      RETURN "UNO"
  ELSE IF N = 2
    THEN
      RETURN "DUE"
  ELSE IF N = 3
    THEN
      RETURN "TRE"
  ELSE IF N = 4
    THEN
      RETURN "QUATTRO"
```

```
  ELSE IF N = 5
    THEN
      RETURN "CINQUE"
  ELSE IF N = 6
    THEN
      RETURN "SEI"
  ELSE IF N = 7
    THEN
      RETURN "SETTE"
  ELSE IF N = 8
    THEN
      RETURN "OTTO"
  ELSE IF N = 9
    THEN
      RETURN "NOVE"
  ELSE IF N = 10
    THEN
      RETURN "DIECI"
  ELSE IF N = 11
    THEN
      RETURN "UNDICI"
  ELSE IF N = 12
    THEN
      RETURN "DODICI"
  ELSE IF N = 13
    THEN
      RETURN "TREDICI"
  ELSE IF N = 14
    THEN
      RETURN "QUATTORDICI"
  ELSE IF N = 15
    THEN
      RETURN "QUINDICI"
  ELSE IF N = 16
```

```
    THEN
      RETURN "SEDICI"
  ELSE IF N = 17
    THEN
      RETURN "DICIASSETTE"
  ELSE IF N = 18
    THEN
      RETURN "DICIOTTO"
  ELSE IF N = 19
    THEN
      RETURN "DICIANNOVE"
  ELSE IF N = 20
    THEN
      RETURN "VENTI"
  ELSE IF N = 21
    THEN
      RETURN "VENTUNO"
  ELSE IF (N >= 22) AND (N <= 29)
    THEN
      RETURN "VENTI" INTEGER_TO_ITALIAN (N-20)
  ELSE IF N = 30
    THEN
      RETURN "TRENTA"
  ELSE IF N = 31
    THEN
      RETURN "TRENTUNO"
  ELSE IF (N >= 32) AND (N <= 39)
    THEN
      RETURN "TRENTA" INTEGER_TO_ITALIAN (N-30)

  …
  …
  ELSE IF N = 90
    THEN
```

```
    RETURN "NOVANTA"
ELSE IF N = 91
  THEN
    RETURN "NOVANTUNO"
ELSE IF (N >= 92) AND (N <= 99)
  THEN
    RETURN "NOVANTA" INTEGER_TO_ITALIAN (N-90)
ELSE IF (N >= 100) AND (N <= 199)
  THEN
    RETURN "CENTO" INTEGER_TO_ITALIAN (N-100)
ELSE IF (N >= 200) AND (N <= 999)
  THEN
    X := int (N / 100)
    Y := N - (X * 100)
    RETURN INTEGER_TO_ITALIAN (X)
           "CENTO"
           INTEGER_TO_ITALIAN (Y)
ELSE IF (N >= 1000) AND (N <= 1999)
  THEN
    RETURN "MILLE" INTEGER_TO_ITALIAN (N-1000)
ELSE IF (N >= 2000) AND (N <= 999999)
  THEN
    X := int (N / 1000)
    Y := N - (X * 1000)
    RETURN INTEGER_TO_ITALIAN (X)
           "MILA"
           INTEGER_TO_ITALIAN (Y)
ELSE IF (N >= 1000000) AND (N <= 1999999)
  THEN
    RETURN "UNMILIONE"
           INTEGER_TO_ITALIAN (N-1000000)
ELSE IF (N >= 2000000) AND (N <= 999999999)
  THEN
    X := int (N / 1000000)
```

```
      Y := N - (X * 1000000)
      RETURN INTEGER_TO_ITALIAN (X)
             "MILIONI"
             INTEGER_TO_ITALIAN (Y)
  ELSE IF (N >= 1000000000) AND (N <= 1999999999)
    THEN
      RETURN "UNMILIARDO"
             INTEGER_TO_ITALIAN (N-1000000000)
  ELSE IF (N >= 2000000000) AND (N <= 999999999999)
    THEN
      X := int (N / 1000000000)
      Y := N - (X * 1000000000)
      RETURN INTEGER_TO_ITALIAN (X)
             "MILIARDI"
             INTEGER_TO_ITALIAN (Y)
  ELSE
    "##ERROR##"
  END IF
END INTEGER_TO_ITALIAN
```

62.8 Algoritmi elementari con la shell POSIX

Questa sezione raccoglie degli esempi di programmazione per lo studio elementare degli algoritmi, realizzati in forma di script per una shell POSIX. Questi esempi sono ottenuti ricostruendo un lavoro didattico del 1983, realizzato allora con degli script ARCS, un linguaggio del sistema operativo CMS (*Computer management system*) Burroughs.

62.8.1 ARCS0: ricerca del valore più grande tra tre numeri interi

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS0.sh* .

```
#!/bin/sh
##
## ARCS0 1983-07-02
##
## Trovare il più grande fra tre numeri interi, diversi tra
## loro.
##

printf "inserisci il primo numero   "
read a

printf "inserisci il secondo numero "
read b

printf "inserisci il terzo numero   "
read c

if [ $a -gt $b ]
then
    if [ $a -gt $c ]
    then
        echo "il numero maggiore è $a"
    else
        echo "il numero maggiore è $c"
    fi
else
    if [ $b -gt $c ]
    then
        echo "il numero maggiore è $b"
    else
```

```
        echo "il numero maggiore è $c"
    fi
fi
```

62.8.2 ARCS1: moltiplicazione di due numeri interi

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS1.sh* .

```
##
## ARCS1 1983-07-06
##
## Moltiplicazione di due numeri.
##

printf "inserisci il primo numero   - 0 per finire "
read x
printf "inserisci il secondo numero - 0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=0
    while [ $y -ne 0 ]
    do
        z=$(($z + $x))
        y=$(($y - 1))
    done
    echo "il risultato è $z"
    printf "inserisci il primo numero   - 0 per finire "
    read x
    printf "inserisci il secondo numero - 0 per finire "
    read y
done
```

62.8.3 ARCS2: valore assoluto della differenza tra due valori

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS2.sh* .

```
#!/bin/sh
##
## ARCS2 1983-07-06
##
## Valore assoluto della differenza tra due numeri.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    if [ $x -lt $y ]
    then
        u=$x
        x=$y
        y=$u
    fi
    z=$(($x - $y))
    echo "|x-y| = $z"
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.4 ARCS3: somma tra due numeri

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS3.sh* .

```
#!/bin/sh
##
## ARCS3 1983-07-07
##
## Somma tra due numeri.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    while [ $y -ne 0 ]
    do
        x=$(($x + 1 ))
        y=$(($y - 1))
    done
    echo "x+y = $x"
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.5 ARCS4: prodotto tra due numeri

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS4.sh* .

```
#!/bin/sh
##
## ARCS4 1983-07-07
##
## Prodotto tra due numeri.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=0
    t=$y
    while [ $t -ne 0 ]
    do
        u=$z
        v=$x
        while [ $v -ne 0 ]
        do
            u=$(($u + 1))
            v=$(($v - 1))
        done
        z=$u
        t=$(($t - 1))
    done
    echo "x*y = $z"
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.6 ARCS5: quoziente

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS5.sh* .

```
#!/bin/sh
##
## ARCS5 1983-07-07
##
## Quoziente.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    if [ $y -eq 0 ]
    then
        echo "x / 0 = indefinito"
    else
        z=0
        if [ $x -ge $y ]
        then
            u=$x
            until [ $u -le 0 ]
            do
                u=$(($u - $y))
                if [ $u -ge 0 ]
                then
                    z=$(($z + 1))
                fi
            done
            echo "x / y = $z"
        else
```

```
            echo "x / y = $z"
        fi
    fi
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.7 ARCS6: verifica della parità di un numero

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS6.sh* .

```
#!/bin/sh
##
## ARCS6 1983-07-07
##
## Verifica della parità di un numero.
##

printf "inserisci x   0 per finire "
read x
until [ $x -eq 0 ]
do
    y=$(($x / 2) * 2))
    if [ $x -ne $y ]
    then
        echo "il numero $x è dispari"
    else
        echo "il numero $x è pari"
    fi
    printf "inserisci x   0 per finire "
    read x
done
```

62.8.8 ARCS7: fattoriale

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS7.sh* .

```
#!/bin/sh
##
## ARCS7 1983-07-08
##
## Fattoriale di un numero.
##

printf "inserisci il numero    99 per finire "
read n
until [ $n -eq 99 ]
do
    z=1
    k=0
    while [ $k -ne $n ]
    do
        k=$(($k + 1))
        z=$(($z * $k))
    done
    echo "$n! = $z"
    printf "inserisci il numero    99 per finire "
    read n
done
```

62.8.9 ARCS8: coefficiente binomiale

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS8.sh* .

```
#!/bin/sh
##
## ARCS8 1983-07-08
```

```
##
## Coefficiente binomiale.
##
## /n\  =  n*(n-1)*...*(n-k+1) / k!
## \k/
##

printf "inserisci n    999 per finire "
read a
printf "inserisci k    999 per finire "
read b
until [ $a -eq 999 ] && [ $b = 999 ]
do
    if [ $b -eq 0 ]
    then
        echo "il coefficiente binomiale di n su 0 è 1"
    else
        k1=0
        z1=1
        while [ $k1 -ne $b ]
        do
            k1=$(($k1 + 1))
            z1=$((z1 * k1))
        done
        y=$z1
        k2=$(($a - $b))
        z2=1
        while [ $k2 -ne $a ]
        do
            k2=$(($k2 + 1))
            z2=$(($z2 * $k2))
        done
        x=$z2
        z=$(($x / $y))
```

```
        echo "il coefficiente binomiale è $x/$y"
        echo "che se viene calcolato con approssimazione"
        echo "di una unità, dà $z"
    fi
    printf "inserisci n    999 per finire "
    read a
    printf "inserisci k    999 per finire "
    read b
done
```

62.8.10 ARCS10: massimo comune divisore

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS10.sh* .

```
#!/bin/sh
##
## ARCS10 1983-07-09
##
## M.C.D.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=$x
    w=$y
    while [ $z -ne $w ]
    do
        if [ $z -gt $w ]
        then
            z=$(($z - $w))
```

```
        else
            w=$(($w - $z))
        fi
    done
    echo "il M.C.D. di $x e $y è $z"
    printf "inserisci x    0 per finire "
    read x
    printf "inserisci y    0 per finire "
    read y
done
```

62.8.11 ARCS11: massimo comune divisore

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS11.sh* .

```
#!/bin/sh
##
## ARCS11 1983-07-09
##
## M.C.D.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=$x
    w=$y
    while [ $z -ne $w ]
    do
        while [ $z -gt $w ]
        do
```

```
            z=$(($z - $w))
        done
        while [ $w -gt $z ]
        do
            w=$(($w - $z))
        done
    done
    echo "il M.C.D. di $x e $y è $z"
    printf "inserisci x    0 per finire "
    read x
    printf "inserisci y    0 per finire "
    read y
done
```

62.8.12 ARCS12: radice quadrata intera

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS12.sh* .

```
#!/bin/sh
##
## ARCS12 1983-07-09
##
## radice quadrata intera
##

printf "inserisci il numero di cui vuoi la radice "
printf "   0 per finire "
read x
until [ $x -eq 0 ]
do
    z=0
    t=0
    until [ $t -ge $x ]
    do
```

```
        z=$(($z + 1))
        t=$(($z * $z))
    done
    if [ $t -ne $x ]
    then
        z=$(($z - 1))
    fi
    echo "la radice intera di $x è $z"
    printf "inserisci il numero di cui vuoi la radice "
    printf "   0 per finire "
    read x
done
```

62.8.13 ARCS13: numero primo

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS13.sh* .

```
#!/bin/sh
##
## ARCS13 1983-07-09
##
## il numero è primo?
##

printf "inserisci il numero     9999 per finire "
read x
until [ $x -eq 9999 ]
do
    primo=1
    k=2
    while [ $k -lt $x ] && [ $primo -eq 1 ]
    do
        t=$(($x / $k))
        t=$(($x - ($t * $k)))
```

```
        if [ $t -eq 0 ]
        then
            primo=0
        else
            k=$(($k + 1))
        fi
    done
    if [ $primo -eq 1 ]
    then
        echo "il numero $x è primo"
    else
        echo "il numero $x non è primo"
    fi
    printf "inserisci il numero     9999 per finire "
    read x
done
```

62.8.14 ARCS14: numero primo

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS14.sh* .

```
#!/bin/sh
##
## ARCS14 1983-07-09
##
## il numero è primo?
## questa versione non funziona correttamente con 0 e 1.
##

printf "inserisci il numero     9999 per finire "
read x
until [ $x -eq 9999 ]
do
    primo=1
```

```
    z=0
    t=0
    until [ $t -ge $x ]
    do
        z=$(($z + 1))
        t=$(($z * $z))
    done
    if [ $t -ne $x ]
    then
        z=$(($z - 1))
    else
        primo=0
    fi
    k=2
    while [ $k -lt $x ] && [ $primo -eq 1 ]
    do
        t=$(($x / $k))
        t=$(($x - ($t * $k)))
        if [ $t -eq 0 ]
        then
            primo=0
        else
            k=$(($k + 1))
        fi
    done
    if [ $primo -eq 1 ]
    then
        echo "il numero $x è primo"
    else
        echo "il numero $x non è primo"
    fi
    printf "inserisci il numero     9999 per finire "
    read x
done
```

62.9 Riferimenti

«

- Th. Estier, *What is BNF notation?*, *http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html*