Esercizio 1

Dati $n, k > 0$, calcolare i primi n multipli di k.

Soluzione

La soluzione del problema è piuttosto elementare: letti in input i dati del problema, si iterano le istruzioni con cui si calcola l'i-esimo multiplo di k (per $i = 1, 2, \ldots, n$) come prodotto $i \times k$ e si stampa il risultato. È una classica applicazione della struttura iterativa, il cui diagramma di flusso è rappresentato in Figura 1.

1: leggi n e k
2: $i = 1$
3: scrivi $i \times k$
4: $i = i + 1$
5: se $i \leq n$ vai al passo 3
6: stop

La complessità computazionale dell'algoritmo è $O(n)$, perché viene reiterato n volte il ciclo rappresentato dalle righe 3–5.

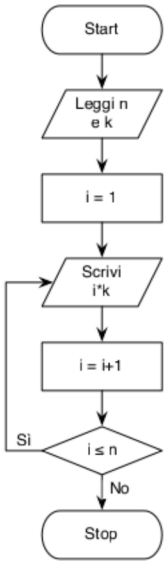

Figura 1: Diagramma di flusso della soluzione dell'Esercizio 1

Di seguito è riportata una codifica in linguaggio C dell'algoritmo che può essere compresa meglio riscrivendo la pseudo-codifica dell'algoritmo in questo modo:

1: leggi n e k
2: $i = 1$
3: **ripeti**
4: scrivi $i \times k$
5: $i = i + 1$
6: **fintanto che** $i \leq n$
7: stop

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  int n, k, i;
```

```
5     printf("Inserisci due numeri interi n e k: ");
6     scanf("%d %d", &n, &k);
7     i = 1;
8     do {
9       printf("%d\n", i*k);
10      i = i+1;
11    } while (i <= n);
12    return(0);
13  }
```

Esercizio 2

Dati $n, k > 0$ stabilire se n è divisibile per k.

Soluzione

La soluzione si basa sull'applicazione di una definizione assai elementare del concetto di divisibilità tra due numeri interi: possiamo dire che n è divisibile per k se k "entra" in n esattamente d volte; in altri termini n è divisibile per k se esiste un intero d tale che $n = kd$. La verifica avviene quindi sottraendo ripetutamente k da n fino ad ottenere il valore 0 nel caso in cui n sia divisibile per k o un numero maggiore di zero e minore di k, nel caso in cui n non sia divisibile per k.

1: leggi n e k
2: $m = n$
3: se $m > 0$ allora prosegui, altrimenti vai al passo 5
4: $m = m - k$
5: vai al passo 3
6: se $m = 0$ allora scrivi "Sì, n è divisibile per k"
7: altrimenti scrivi "No, n non è divisibile per k"
8: stop

Anche in questo caso, come evidenziato nel diagramma di flusso in Figura 2, viene utilizzata una struttura iterativa (righe 2–4); al termine della struttura iterativa viene usata una struttura condizionale (righe 5–6) per verificare il risultato dell'elaborazione effettuata mediante il ciclo.

La complessità computazionale dell'algoritmo è $O(n/k)$, perché il ciclo delle righe 2–4 viene reiterato proprio n/k volte.

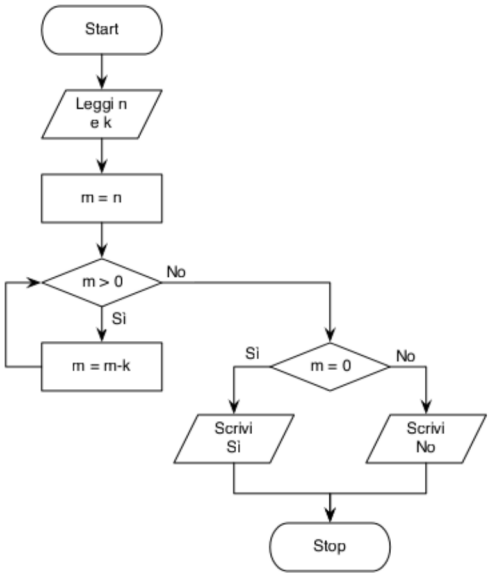

Figura 2: Diagramma di flusso della soluzione dell'Esercizio 2

Di seguito riportiamo una codifica in linguaggio C dell'algoritmo. Anche in questo caso il programma C può essere compreso più facilmente riscrivendo l'algoritmo in modo tale da evidenziare le strutture algoritmiche iterativa e condizionale:

1: leggi n e k
2: $m = n$

3: **fintanto che** $m \geq k$ **ripeti**
4: $m = m - k$
5: **fine-ciclo**
6: **se** $m = 0$ **allora**
7: scrivi "Sì, n è divisibile per k"
8: **altrimenti**
9: scrivi "No, n non è divisibile per k"
10: **fine-condizione**
11: stop

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  int n, k, m;
  printf("Inserisci due interi n e k: ");
  scanf("%d %d", &n, &k);
  m = n;
  while (m >= k)
    m = m-k;
  if (m == 0)
    printf("Si', %d e' divisibile per %d.\n", n, k);
  else
    printf("No, %d non e' divisibile per %d.\n", n, k);
  return(0);
}
```

In questo caso il ciclo è stato implementato utilizzando l'istruzione `while`, visto che la condizione deve essere valutata ogni volta prima di ripetere il ciclo e non al termine del ciclo stesso; inoltre, poiché sia il blocco di istruzioni del ciclo `while` che i blocchi di istruzioni della condizione `if` e dell'alternativa `else` sono costituite da una sola istruzione, vengono omesse le parentesi graffe che delimitano i blocchi.

Esercizio 3

Dato $n > 1$ stabilire se è un numero primo.

Soluzione

La soluzione di questo problema è basata sulla soluzione del problema precedente: n è primo se non è divisibile per nessuno degli interi k compresi tra 1 ed n (esclusi gli estremi: $1 < k < n$).

La soluzione si ottiene quindi iterando per $k = 2, 3, \ldots, n-1$ l'algoritmo con cui si risolve il problema "stabilire se n è divisibile per k". A sua volta questo algoritmo, come abbiamo visto nelle pagine precedenti, è basato su una struttura iterativa (righe 5–7) seguita da una struttura condizionale (riga 8); entrambe queste strutture saranno inserite all'interno di una struttura iterativa (righe 4–9) con cui si fa variare il valore di k da 2 fino a $n-1$ o fino a quando non sia stato individuato un valore di k minore di n tale che n sia divisibile per k. Al termine di questa struttura iterativa con una struttura condizionale (righe 10–11) si verifica l'esito dell'elaborazione e si stampa il risultato.

1: leggi n
2: $r = 0$
3: $k = 1$
4: $m = n$, $k = k + 1$
5: se $m > 0$ allora prosegui, altrimenti vai al passo 8
6: $m = m - k$
7: vai al passo 5
8: se $m = 0$ allora $r = 1$
9: se $r = 0$ e $k < n - 1$ allora vai al passo 4
10: se $r = 1$ allora scrivi "n non è un numero primo (è divisibile per k)"
11: altrimenti scrivi "n è un numero primo"
12: stop

La complessità computazionale dell'algoritmo nel caso peggiore è data dal numero di operazioni effettuate nel caso in cui n sia un numero primo; in tal caso infatti il ciclo più esterno, definito alle righe 4–9, viene eseguito il massimo numero di volte ($k = 2, \ldots, n-1$); per ciascun ciclo (e per ciascun valore di $k = 2, \ldots, n-1$) il ciclo definito alle righe 5–7 viene eseguito n/k volte (vedi l'esercizio 2), per stabilire se n è divisibile per k. Complessivamente vengono quindi eseguite $n/2 + n/3 + \ldots n/(n-1)$ operazioni.

Lo stesso algoritmo può essere ridefinito evidenziando meglio le strutture di controllo iterative e condizionali con la seguente pseudo-codifica in cui non sono riportate esplicitamente le istruzioni di "salto" per evidenziare che si tratta sempre di istruzioni di "salto condizionato", così come richiesto dalle regole della programmazione strutturata.

1: leggi n
2: $r = 0$
3: $k = 1$
4: **ripeti**
5: $m = n$, $k = k + 1$
6: **fintanto che** $m > 0$ **ripeti**
7: $m = m - k$
8: **fine-ciclo**
9: **se** $m = 0$ **allora**
10: $r = 1$
11: **fine-condizione**
12: **fintanto che** $r = 0$ e $k < n - 1$
13: **se** $r = 1$ **allora**
14: scrivi "n non è un numero primo (è divisibile per k)"

15: **altrimenti**
16: scrivi "n è un numero primo"
17: **fine-condizione**
18: stop

Di seguito riportiamo il diagramma di flusso con cui viene codificato l'algoritmo; naturalmente il diagramma di flusso è identico sia nel caso della prima versione dell'algoritmo che della seconda, dal momento che le due pseudo-codifiche sono due modi equivalenti di descrivere il medesimo procedimento risolutivo. Nei blocchi punteggiati sono evidenziate le strutture di controllo nidificate una dentro l'altra o concatenate una di seguito all'altra nel rispetto delle regole della programmazione strutturata.

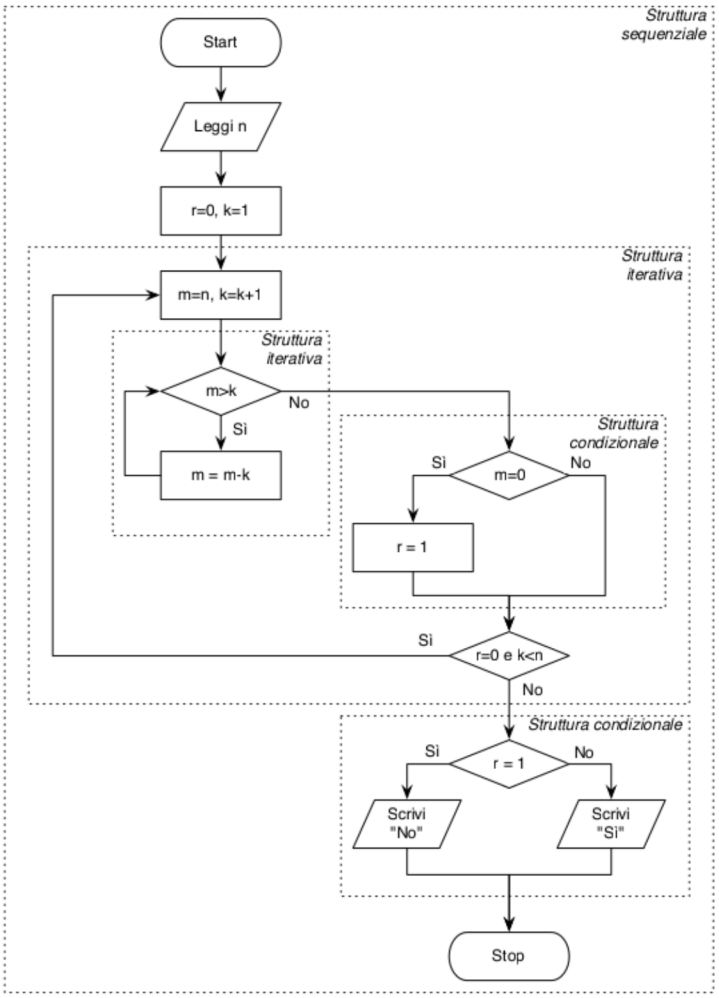

Figura 3: Diagramma di flusso della soluzione dell'Esercizio 3

Infine riportiamo la codifica in linguaggio C del programma che implementa l'algoritmo.

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  int n, k, r, m;
  printf("Inserisci un intero maggiore di zero: ");
  scanf("%d", &n);
```

```
  r = 0;
  k = 1;
  do {
    m = n;
    k = k+1;
    while (m > 0) {
      m = m-k;
    }
    if (m == 0)
      r = 1;
  } while (r == 0 && k < n-1);
  if (r == 1)
    printf("%d non e' primo (e' divisibile per %d).\n", n, k);
  else
    printf("%d e' un numero primo.\n", n);
  return(0);
}
```

Lo stesso programma può essere codificato in maniera differente, realizzando una funzione booleana chiamata `divisibile` che opera su due numeri interi a e b e che restituisce "vero" (in C un valore intero diverso da zero) nel caso in cui a sia divisibile per b e "falso" (in C il valore zero) in caso contrario. Tale funzione deve implementare le istruzioni che nella codifica precedente sono riportate alle righe di programma 11–16. Riportiamo di seguito la seconda versione del programma in linguaggio C.

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

int divisibile(int a, int b) {
  int risp;
  while (a > 0)
    a = a-b;
  if (a == 0)
    risp = 1;
  else
    risp = 0;
  return(risp);
}

int main(void) {
  int n, k, r;
  printf("Inserisci un intero maggiore di zero: ");
  scanf("%d", &n);
  r = 0;
  k = 1;
  do {
    k = k+1;
    if (divisibile(n, k))
      r = 1;
  } while (r == 0 && k < n-1);
  if (r == 1)
    printf("%d non e' primo (e' divisibile per %d).\n", n, k);
  else
    printf("%d e' un numero primo.\n", n);
  return(0);
}
```

Esercizio 4

Dati due interi x e y tali che $x > y > 0$, calcolare la parte intera e il resto della divisione x/y.

Soluzione

La strategia risolutiva per questo problema sfrutta ancora una volta la procedura per la verifica della divisibilità vista nell'esercizio 2 e riutilizzata anche nell'esercizio 3. Infatti il rapporto x/y è uguale a d con resto r $(0 \leq r < y)$ se $x = y \cdot d + r$. In termini più rudimentali possiamo dire che d, la parte intera del risultato della divisione, è dato da quante volte y "entra" in x; mentre il resto r è dato dal valore che "avanza" dopo aver tolto per d volte y da x. Possiamo proporre quindi la seguente pseudo-codifica:

1: leggi x e y
2: $r = n$, $d = 0$
3: se $r \geq y$ allora prosegui, altrimenti vai al passo 5
4: $r = r - y$, $d = d + 1$
5: vai al passo 3
6: scrivi "$x/y = d$ con resto r"
7: stop

La pseudo-codifica può essere riscritta mettendo in evidenza i salti condizionati e la struttura iterativa dell'algoritmo:

1: leggi x e y
2: $r = n$, $d = 0$
3: **fintanto che** $r \geq y$ **ripeti**
4: $\quad r = r - y$, $d = d + 1$
5: **fine-ciclo**
6: scrivi "$x/y = d$ con resto r"
7: stop

È abbastanza evidente che il numero di operazioni elementari svolte dall'algoritmo è pari al numero di volte che viene iterato il ciclo alle righe 3–5; dunque la complessità dell'algoritmo è $O(x/y)$.

Riportiamo di seguito il diagramma di flusso e la codifica in linguaggio C dell'algoritmo.

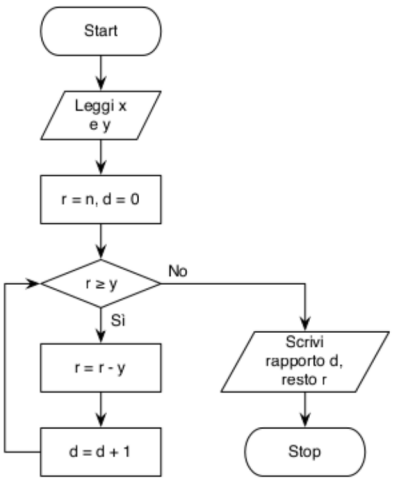

Figura 4: Diagramma di flusso della soluzione dell'Esercizio 4

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  int x, y, r, d;
  printf("inserisci due interi x e y (x>y): ");
  scanf("%d %d", &x, &y);
  d = 0;
  r = x;
  while (r >= y) {
    r = r-y;
    d = d+1;
  }
  printf("rapporto: %d\nresto: %d\n", d, r);
  return(0);
}
```

Per evitare malintesi è bene precisare che questo esercizio tende a mettere in evidenza la possibilità di ottenere il risultato richiesto utilizzando soltanto degli operatori aritmetici di base. Tuttavia in linguaggio C la divisione fra due variabili intere restituisce sempre un resto intero (a meno di non utilizzare il *cast*) e il resto della divisione intera fra due numeri può essere ottenuto mediante l'operatore di *modulo* indicato con il simbolo "%". Possiamo quindi proporre la seguente codifica in C più compatta della precedente:

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
  int x, y;
  printf("inserisci due interi x e y (x>y): ");
  scanf("%d %d", &x, &y);
  printf("rapporto: %d\nresto: %d\n", x/y, x%y);
  return(0);
}
```

Esercizio 5

Dati due numeri interi positivi x e y calcolare il massimo comune divisore tra x e y.

Soluzione

Per costruire un algoritmo elementare per calcolare il MCD (massimo comun divisore) tra x e y ancora una volta facciamo ricorso ad una divisione piuttosto elementare in modo da costruire la strategia risolutiva basandoci soltanto su operazioni aritmetiche di base. In particolare l'idea è quella di applicare una variante del cosiddetto "algoritmo di Euclide": si tratta di sottrarre ripetutamente x da y (se $y > x$) o y da x (se $x > y$) fino ad ottenere lo stesso valore per x e y; tale valore è proprio il massimo comun divisore dei due numeri originali.

La strategia risolutiva appena descritta può essere precisata con il seguente algoritmo di cui riportiamo la pseudo-codifica:

1: leggi x e y
2: se $x \neq y$ prosegui, altrimenti vai al passo 6
3: se $x > y$ allora $x = x - y$
4: altrimenti $y = y - x$
5: vai al passo 2
6: scrivi "il MCD è x"
7: stop

Come al solito possiamo riscrivere la pseudo-codifica per evidenziare meglio le strutture algoritmiche iterativa e condizionale.

1: leggi x e y
2: **fintanto che** $x \neq y$ **ripeti**
3: **se** $x > y$ **allora**
4: $x = x - y$
5: **altrimenti**
6: $y = y - x$
7: **fine-condizione**
8: **fine-ciclo**
9: scrivi "il MCD è x"
10: stop

L'algoritmo può essere codificato in un programma C come segue.

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

int MCD(int x, int y) {
  while (x != y)
    if (x > y)
      x = x - y;
    else
      y = y - x;
  return(x);
}

int main(void) {
  int a, b;
  printf("Inserisci due numeri interi: ");
  scanf("%d %d", &a, &b);
  printf("Il massimo comune divisore e' %d\n", MCD(a,b));
  return(0);
}
```

Riportiamo infine la rappresentazione dell'algoritmo con un diagramma di flusso in cui risulta evidente che la struttura iterativa del ciclo rappresentato con le righe 2–5 della prima pseudo-codifica o con le righe 2–8 della seconda, contiene una struttura condizionale (righe 3–4 e righe 3–7 rispettivamente nella prima e nella seconda versione della pseudo-codifica).

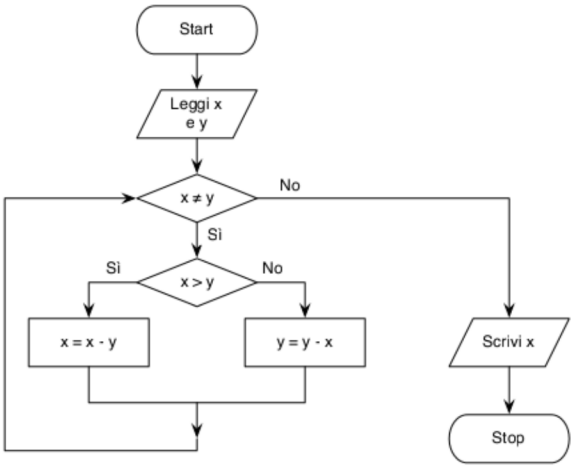

Figura 5: Diagramma di flusso della soluzione dell'Esercizio 5

Esercizio 6

Dati due numeri interi positivi x e y calcolare il minimo comune multiplo tra x e y.

Soluzione

La strategia risolutiva è la seguente: siano $m_x = x$ ed $m_y = y$ i primi multipli di x e y rispettivamente; se $m_x < m_y$ si somma x ad m_x ottenendo il successivo multiplo di x, altrimenti si somma y ad m_y ottenendo il successivo multiplo di y; il procedimento viene iterato fino a quando non si ottengono due valori uguali per m_x e m_y. Naturalmente il procedimento ha termine dopo un numero finito di passi; infatti bisogna trovare due interi $h, k > 0$ tali che $hx = ky$. Il procedimento ha quindi una complessità di $O(h+k)$: si deve sommare h volte x e k volte y prima di individuare il più piccolo multiplo comune ad entrambi.

La seguente pseudo-codifica formalizza in un algoritmo la strategia risolutiva sopra esposta.

1: leggi x e y
2: $m_x = x$, $m_y = y$
3: se $m_x \neq m_y$ prosegui, altrimenti vai al passo 7
4: se $m_x < m_y$ allora $m_x = m_x + x$
5: altrimenti $m_y = m_y + y$
6: vai al passo 3
7: scrivi "il mcm tra x e y è m_x"
8: stop

Ancora una volta riscriviamo la pseudo codifica in modo da mettere in evidenza le strutture algoritmiche utilizzate.

1: leggi x e y
2: $m_x = x$, $m_y = y$
3: **fintanto che** $m_x \neq m_y$ **ripeti**
4: **se** $m_x < m_y$ **allora**
5: $m_x = m_x + x$
6: **altrimenti**
7: $m_y = m_y + y$
8: **fine-condizione**
9: **fine-ciclo**
10: scrivi "il mcm tra x e y è m_x"
11: stop

L'algoritmo può essere codificato in un programma C come segue. Come nell'esercizio precedente l'algoritmo per il calcolo del minimo comune multiplo è codificato in una funzione richiamata dalla funzione principale `main`. In questo modo, se fosse necessario, sarebbe possibile riutilizzare la stessa funzione `mcm` anche nell'ambito di altri programmi.

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

int mcm(int x, int y) {
  int mx = x, my = y;
  while (mx !=m y)
    if (mx < my)
      mx = mx + x;
    else
      my = my + y;
  return(mx);
}
```

```
13
14 int main(void) {
15   int a, b;
16   printf("Inserisci due numeri interi: ");
17   scanf("%d %d", &a, &b);
18   printf("Il minimo comune multiplo e' %d\n", mcm(a,b));
19   return(0);
20 }
```

Di seguito riportiamo il diagramma di flusso dell'algoritmo per il calcolo del minimo comune multiplo tra i due interi x e y.

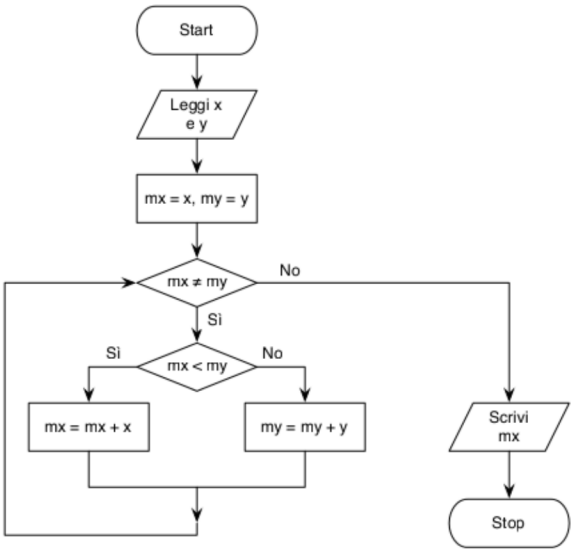

Figura 6: Diagramma di flusso della soluzione dell'Esercizio 6

Esercizio 7

Letti n numeri razionali stabilire se sono stati acquisiti in ordine crescente.

Soluzione

Il problema in sé è piuttosto semplice, ma ci permette di evidenziare l'uso di variabili "flag" che servono a segnalare il fatto che durante l'esecuzione di un procedimento iterativo si è verificata una determinata condizione che non potremo controllare nuovamente al termine del ciclo a meno di non utilizzare questa variabile che ci permetterà di stabilire se la condizione desiderata si era verificata o meno.

Il problema richiede infatti di iterare un ciclo con il quale vengono acquisiti in input gli n numeri razionali; man mano che questi numeri vengono letti dovremo confrontare l'elemento appena letto con il precedente per stabilire se l'ordine crescente è rispettato o meno. Il procedimento di lettura deve comunque essere portato a termine per tutti gli n elementi dell'insieme, ma si deve tenere traccia del fatto che tutti gli elementi letti in input hanno rispettato l'ordine crescente oppure, al contrario, che almeno una coppia di elementi non rispettava tale ordine. A tal fine possiamo utilizzare una variabile che chiameremo *flag* a cui assegneremo inizialmente il valore 0; durante il processo di lettura degli n elementi se ne incontreremo uno minore del suo predecessore, allora modificheremo il valore di *flag* impostandolo uguale a 1, in caso contrario non modificheremo il suo valore. In questo modo, al termine del procedimento iterativo di lettura, se la variabile *flag* sarà rimasta uguale a zero, allora vorrà dire che tutti gli elementi rispettavano l'ordine crescente; in caso contrario, se $flag = 1$, allora almeno una coppia di elementi non rispettavano l'ordine crescente. Di seguito proponiamo la pseudo-codifica dell'algoritmo.

1: leggi n
2: $flag = 0$
3: leggi a
4: $i = 1$
5: se $i < n$ allora prosegui, altrimenti vai al passo 11
6: leggi b
7: se $b \leq a$ allora $flag = 1$
8: $a = b$
9: $i = i + 1$
10: vai al passo 5
11: se $flag = 0$ allora scrivi "Gli elementi sono in ordine crescente"
12: altrimenti scrivi "Gli elementi non sono in ordine crescente"
13: stop

Riscriviamo la pseudo-codifica mettendo in evidenza il ciclo iterativo principale che a sua volta contiene una struttura condizionale; la struttura iterativa è seguita da una struttura condizionale. La concatenazione e la nidificazione delle strutture algoritmiche è messa in maggiore evidenza dalla rappresentazione grafica dell'algoritmo riportata nel diagramma di flusso di Figura 7.

1: leggi n
2: $flag = 0$
3: leggi a
4: $i = 1$
5: **fintanto che** $i < n$ **ripeti**
6: leggi b
7: **se** $b \leq a$ **allora**
8: $flag = 1$
9: **fine-condizione**

10: $a = b$
11: $i = i + 1$
12: **fine-ciclo**
13: **se** $flag = 0$ **allora**
14: scrivi “Gli elementi sono in ordine crescente”
15: **altrimenti**
16: scrivi “Gli elementi non sono in ordine crescente”
17: **fine-condizione**
18: stop

Riportiamo la codifica in linguaggio C dell’algoritmo. Gli unici due aspetti degni di rilievo sono l’utilizzo dell’istruzione `for` per implementare la struttura di controllo iterativa e l’uso della sequenza “`%f`” nella stringa di formato della funzione `scanf` alle righe 10 e 12 del programma, dal momento che le variabili `a` e `b` erano di tipo `float` e non `int` come per le altre variabili utilizzate fino ad ora.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  int main(void) {
4    int n, flag, i;
5    float a, b;
6    printf("Inserisci il numero di elementi: ");
7    scanf("%d", &n);
8    printf("inserisci %d elementi: ", n);
9    flag = 0;
10   scanf("%f", &a);
11   for (i=1; i<n; i++) {
12     scanf("%f", &b);
13     if (b < a)
14       flag = 1;
15     a = b;
16   }
17   if (flag == 0)
18     printf("Gli elementi sono in ordine crescente.\n");
19   else
20     printf("Gli elementi non sono tutti in ordine crescente.\n");
21   return(0);
22 }
```

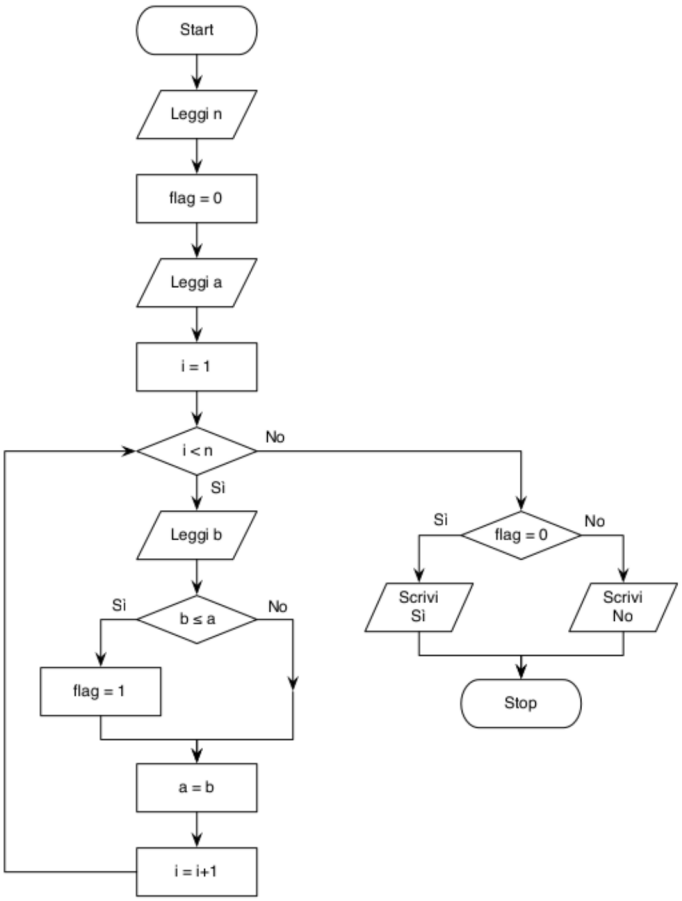

Figura 7: Diagramma di flusso della soluzione dell'Esercizio 7