

Appunti introduttivi sulla progettazione degli algoritmi

Marco Liverani*

Ottobre 2007

Nelle pagine seguenti sono riportati alcuni problemi e le relative soluzioni costituite da semplici algoritmi espressi con una “pseudo-codifica” in linguaggio naturale. Per ogni algoritmo risolutivo si riporta anche un diagramma di flusso e la complessità computazionale nel caso peggiore. Viene anche proposta una codifica in linguaggio C dell’algoritmo, in modo da poter verificare immediatamente il modo con cui le strutture di controllo algoritmiche vengono tradotte nell’ambito di un programma scritto con un linguaggio di programmazione procedurale.

Nella prima sezione di questa breve nota, sono riportati in estrema sintesi i concetti fondamentali che devono essere compresi per poter iniziare la progettazione di un algoritmo e la valutazione della sua complessità computazionale. Per ciascuno di tali argomenti sono stati scritti capitoli interi di libri di informatica teorica e di programmazione, dunque è a tali volumi (alcuni dei quali sono citati in bibliografia) che si rimanda per il necessario approfondimento di quanto viene qui riportato sotto forma di appunti.

Algoritmi e diagrammi di flusso

Ricordiamo che un *algoritmo* è una procedura di calcolo costituita da un numero finito di passi elementari e che termina, producendo la soluzione del problema, dopo aver effettuato un numero finito di operazioni. Con “passi elementari” si intendono delle istruzioni che facciano diretto riferimento alle capacità di base dell’esecutore automatico a cui viene fornito l’algoritmo per risolvere il problema. Tale esecutore automatico, come è noto, oltre a saper eseguire le quattro operazioni aritmetiche (somma, sottrazione, divisione e moltiplicazione) è in grado di memorizzare i dati (tipicamente dei numeri, negli esercizi proposti nelle pagine seguenti) in alcune variabili di memoria identificate da un nome e di eseguire le seguenti sei istruzioni fondamentali che possiamo ritrovare in tutti i linguaggi di programmazione “imperativi/procedurali”:

Algoritmo

assegna: consente di attribuire ad una variabile un valore costante, il valore memorizzato in un’altra variabile o il risultato di un’espressione aritmetica tra valori costanti o valori contenuti in altre variabili (es.: “ $a = 3$ ”, “ $a = b + 3c$ ”, ecc.);

leggi: consente all’esecutore di acquisire dall’esterno un’informazione e di memorizzarla in una variabile di memoria (es.: “leggi a ”);

scrivi: consente all’esecutore di visualizzare/stampare all’esterno un dato costante, un valore memorizzato in una variabile o il risultato di un’espressione aritmetica che coinvolga valori costanti o variabili (es.: “scrivi S ”, “scrivi 15”, “scrivi a ”, “scrivi $a + 7b$ ”, ecc.);

Istruzioni elementari
per l’esecutore degli
algoritmi

* Università Roma Tre, Corso di Laurea in Matematica, E-mail: liverani@mat.uniroma3.it; ultima modifica: 20 novembre 2014

se... allora... altrimenti: consente all'esecutore di valutare un'espressione *booleana* (una condizione logica) il cui valore può essere solo "vero" o "falso"; in base al risultato della valutazione della condizione, vengono eseguite delle istruzioni oppure, in caso contrario, delle altre (es.: "se $a > 0$ allora $a = a - 1$ altrimenti $a = 20$ ");

vai al passo ...: consente di saltare ad una specifica istruzione dell'algoritmo anziché eseguire l'istruzione successiva nella sequenza naturale;

stop: termina l'esecuzione dell'algoritmo.

Ad ognuna delle precedenti istruzioni elementari corrisponde un simbolo con cui è possibile costruire una "rappresentazione grafica" dell'algoritmo, ossia un *diagramma di flusso*. Ogni diagramma di flusso ha un unico punto di inizio ed un unico punto terminale: entrambe queste istruzioni (*start* e *stop*) sono rappresentate da un'ellisse; nella *pseudo-codifica* di un algoritmo espressa come una sequenza di istruzioni elementari l'istruzione *start* è implicita visto che l'algoritmo inizia sempre dal primo passo della pseudo-codifica. Le istruzioni di assegnazione, in generale le più frequenti negli algoritmi, sono rappresentate all'interno di un rettangolo, mentre le operazioni di *input* e di *output* (*leggi* e *scrivi*) sono rappresentate con dei parallelogrammi. Le condizioni sono espresse all'interno di rombi ed infine i salti (*vai al passo...*) sono rappresentati con delle frecce.

Diagrammi di flusso

I box del diagramma di flusso sono collegati fra loro da frecce, ad indicare che nella sequenza delle operazioni è implicito il fatto che dopo aver eseguito una determinata istruzione, a meno di diverse indicazioni rese esplicite nell'algoritmo tramite l'istruzione "*vai al passo...*", deve essere eseguita l'istruzione subito successiva. Naturalmente in ogni box del diagramma di flusso possono entrare una o più frecce e l'unico box in cui non entra nessuna freccia è quello con l'istruzione *start* che indica il punto di inizio dell'algoritmo. Analogamente possiamo dire che da ogni box esce sempre una ed una sola freccia; fanno eccezione soltanto i box per l'istruzione *stop*, da cui non esce alcuna freccia, e per le condizioni, da cui necessariamente devono sempre uscire esattamente due frecce: una per indicare la strada seguita dall'algoritmo nel caso in cui la condizione risulti essere vera e l'altra nel caso in cui la condizione risulti falsa.

Regole di base per la costruzione di diagrammi di flusso

Le strutture algoritmiche con cui si possono costruire i diagrammi di flusso sono soltanto tre, così come previsto dalle regole della programmazione strutturata:

Strutture algoritmiche ammesse dalle regole della programmazione strutturata

1. la struttura *sequenziale*, in cui le istruzioni si susseguono una dopo l'altra senza alcuna variazione al flusso dell'algoritmo;
2. la struttura *condizionale* (o *alternativa*), in cui a fronte di una condizione il flusso dell'algoritmo si biforca per poi ricongiungersi più avanti;
3. la struttura *iterativa*, in cui una sequenza di istruzioni viene ripetuta più volte fintanto che risulta verificata una determinata condizione; la condizione può essere valutata all'inizio della sequenza di istruzioni da ripetere, oppure al termine della sequenza, a seconda dei casi e delle necessità; quando la condizione risulta falsa, il flusso dell'algoritmo prosegue con la prima istruzione esterna alla sequenza da ripetere.

Le tre strutture possono essere *concatenate* fra loro oppure possono essere *nidificate* una dentro l'altra; le tre strutture non si devono invece mai "accavallare": in tal caso si finisce per progettare un algoritmo non elegante e che risulterebbe assai difficile codificare con le istruzioni di un linguaggio di programmazione strutturato, come il linguaggio C o il linguaggio Java. D'altra parte il Teorema Fondamentale della Programmazione Strutturata (Jacopini e Böhm) garantisce che qualunque algoritmo scritto senza rispettare le regole della programmazione strutturata può anche essere ridefinito attenendosi strettamente a tali regole.

Un aspetto particolarmente significativo della programmazione strutturata è costituito dal divieto di utilizzare i cosiddetti “salti incondizionati”, ossia istruzioni del tipo “vai al passo ...” non soggette al controllo diretto di una condizione booleana. Nei linguaggi di programmazione strutturata non si utilizza l’istruzione di salto “go-to”, ma si inserisce il salto sempre all’interno di una struttura iterativa (al termine o all’inizio del ciclo, in base alla valutazione di una condizione si stabilisce se ripetere il blocco di istruzioni oppure se “saltare” ad un’altra istruzione uscendo dalla struttura di controllo iterativa) o condizionale (in base alla valutazione di una condizione si stabilisce se eseguire determinate istruzioni o “saltare” ad altre istruzioni).

Salti condizionati ed incondizionati

Ricordiamo infine che, nel valutare l’efficienza di un algoritmo, si calcola il numero di operazioni elementari eseguite dall’algoritmo stesso a fronte di un insieme di informazioni da elaborare di cardinalità n . La valutazione dell’efficienza di un algoritmo è quindi una funzione f , tipicamente espressa in funzione della variabile n che rappresenta il numero di dati con cui è definita l’istanza del problema da risolvere. Tale funzione viene chiamata *funzione di complessità computazionale dell’algoritmo*. Se $f(n)$ è la complessità dell’algoritmo \mathcal{A} , allora per risolvere un’istanza del problema caratterizzata da $n = 15$ informazioni, l’algoritmo esegue un numero di operazioni pari a $f(15)$. È facile rendersi conto che la funzione con cui si esprime la complessità di un algoritmo è una funzione positiva (un algoritmo non può eseguire un numero negativo di operazioni!) ed in generale è non decrescente: con l’aumentare delle informazioni da elaborare le operazioni effettuate dall’algoritmo non possono diminuire.

Valutazione dell’efficienza di un algoritmo: complessità computazionale

Generalmente non tutte le istanze di un determinato problema, a parità di informazioni da elaborare, sono equivalenti dal punto di vista della complessità computazionale dell’algoritmo risolutivo; esistono istanze “più facili” ed altre “più difficili”. Nel valutare la complessità degli algoritmi, quindi, si considera in genere la complessità nel caso peggiore, ossia la complessità ottenuta considerando l’istanza del problema meno favorevole al nostro procedimento risolutivo, dando per scontato che negli altri casi l’algoritmo non potrà che comportarsi meglio, in modo più efficiente.

Valutazione delle “istanze meno favorevoli” per una determinata strategia risolutiva

Per poter confrontare la complessità di algoritmi differenti, senza “corrompere” il confronto con dettagli tecnici poco significativi o ottimizzazioni di minor conto, si raggruppano le funzioni di complessità in *classi di complessità* espresse mediante la notazione “O grande”:

Classi di complessità computazionale; la notazione “O grande”

$$O(f(n)) = \{g(n) : \exists n_0 > 0, c > 0 \text{ tali che } 0 < g(n) < cf(n) \forall n > n_0\}$$

L’insieme $O(f(n))$ è costituito da numerose (infinite) funzioni $g(n)$ tali da rispettare *definitivamente*, ossia da un certo valore n_0 in poi, la condizione $0 < g(n) < cf(n)$. In pratica l’insieme $O(f(n))$ contiene tutte le funzioni $g(n)$ il cui andamento asintotico (quindi al crescere di n) è dello stesso ordine di grandezza o inferiore a quello di $f(n)$.

Ad esempio se $f(n) = n^2$ allora $O(f(n))$ conterrà le funzioni $g(n) = n^2$, $g(n) = n$, $g(n) = 10n^2 + 5$, $g(n) = n \log n$, $g(n) = k$, ecc. Viceversa, l’insieme $O(n^2)$ non contiene, ad esempio, la funzione n^3 . È interessante osservare che $O(n) \subset O(n^2) \subset O(n^3) \subset \dots \subset O(2^n) \subset O(n!) \subset \dots$

Nel valutare la complessità di un algoritmo si cercherà di individuare la classe di complessità più bassa che contenga la funzione con cui si può esprimere il numero di operazioni elementari eseguite dall’algoritmo nel caso peggiore. È chiaro infatti che se un algoritmo ha una complessità appartenente alla classe $O(n)$, in generale sarà più efficiente di un algoritmo che, pur risolvendo correttamente il medesimo problema, appartenga alla classe di complessità $O(n^3)$.

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 $i = 1, 2, \dots, n$) 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.

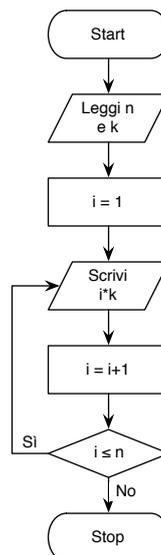


Figura 1: Diagramma di flusso della soluzione dell'Esercizio 1

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

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void) {
4     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 6
- 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.

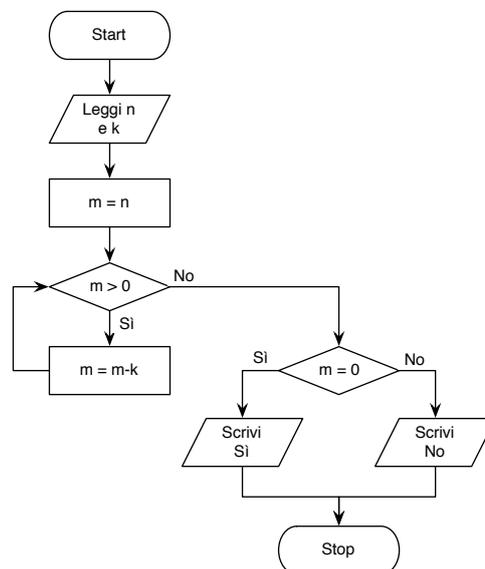


Figura 2: Diagramma di flusso della soluzione dell'Esercizio 2

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 iterative e condizionale:

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

3: **fin tanto 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

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void) {
4     int n, k, m;
5     printf("Inserisci due interi n e k: ");
6     scanf("%d %d", &n, &k);
7     m = n;
8     while (m >= k)
9         m = m - k;
10    if (m == 0)
11        printf("Sì, %d e' divisibile per %d.\n", n, k);
12    else
13        printf("No, %d non e' divisibile per %d.\n", n, k);
14    return(0);
15 }

```

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 $k = 2, 3, \dots, n - 1$ 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 ($k = 2, \dots, n - 1$); per ciascun ciclo (e per ciascun valore di $k = 2, \dots, n - 1$) 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 $n/2 + n/3 + \dots + n/(n - 1)$ 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.

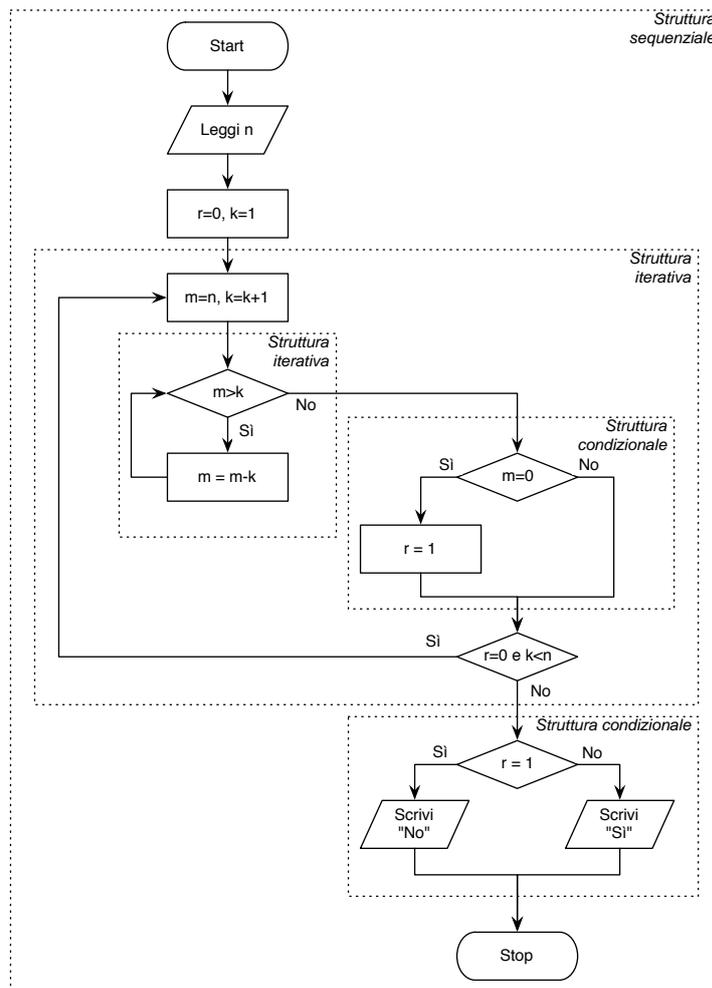


Figura 3: Diagramma di flusso della soluzione dell'Esercizio 3

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

```

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

```

7   r = 0;
8   k = 1;
9   do {
10    m = n;
11    k = k+1;
12    while (m > 0) {
13        m = m-k;
14    }
15    if (m == 0)
16        r = 1;
17    } while (r == 0 && k < n-1);
18    if (r == 1)
19        printf("%d non e' primo (e' divisibile per %d).\n", n, k);
20    else
21        printf("%d e' un numero primo.\n", n);
22    return(0);
23 }

```

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.

```

1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int divisibile(int a, int b) {
5       int risp;
6       while (a > 0)
7           a = a-b;
8       if (a == 0)
9           risp = 1;
10      else
11          risp = 0;
12      return(risp);
13  }
14
15  int main(void) {
16      int n, k, r;
17      printf("Inserisci un intero maggiore di zero: ");
18      scanf("%d", &n);
19      r = 0;
20      k = 1;
21      do {
22          k = k+1;
23          if (divisibile(n, k))
24              r = 1;
25      } while (r == 0 && k < n-1);
26      if (r == 1)
27          printf("%d non e' primo (e' divisibile per %d).\n", n, k);
28      else
29          printf("%d e' un numero primo.\n", n);
30      return(0);
31  }

```

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: $r = r - y$, $d = d + 1$
- 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.

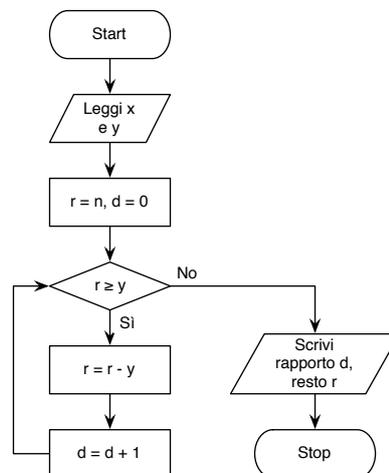


Figura 4: Diagramma di flusso della soluzione dell'Esercizio 4

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void) {
4     int x, y, r, d;
5     printf("inserisci due interi x e y (x>y): ");
6     scanf("%d %d", &x, &y);
7     d = 0;
8     r = x;
9     while (r >= y) {
10        r = r-y;
11        d = d+1;
12    }
13    printf("rapporto: %d\nresto: %d\n", d, r);
14    return(0);
15 }

```

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:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void) {
4     int x, y;
5     printf("inserisci due interi x e y (x>y): ");
6     scanf("%d %d", &x, &y);
7     printf("rapporto: %d\nresto: %d\n", x/y, x%y);
8     return(0);
9 }

```

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.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int MCD(int x, int y) {
5     while (x != y)
6         if (x > y)
7             x = x - y;
8         else
9             y = y - x;
10    return(x);
11 }
12
13 int main(void) {
14     int a, b;
15     printf("Inserisci due numeri interi: ");
16     scanf("%d %d", &a, &b);
17     printf("Il massimo comune divisore e' %d\n", MCD(a,b));
18     return(0);
19 }
```

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).

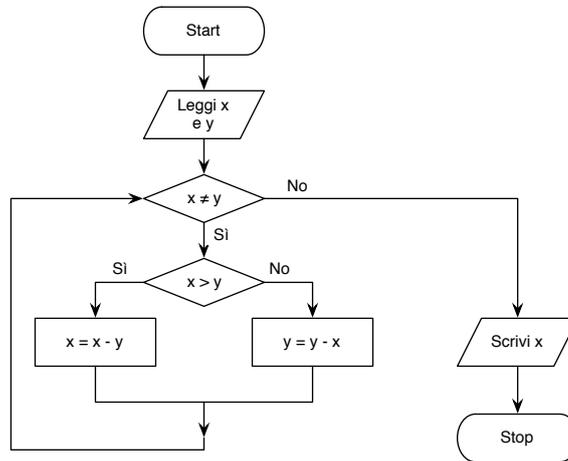


Figura 5: Diagramma di flusso della soluzione dell'Esercizio 5

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: **fi**ntanto 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.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int mcm(int x, int y) {
5     int mx = x, my = y;
6     while (mx != my)
7         if (mx < my)
8             mx = mx + x;
9         else
10            my = my + y;
11     return(mx);
12 }
```

```

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 .

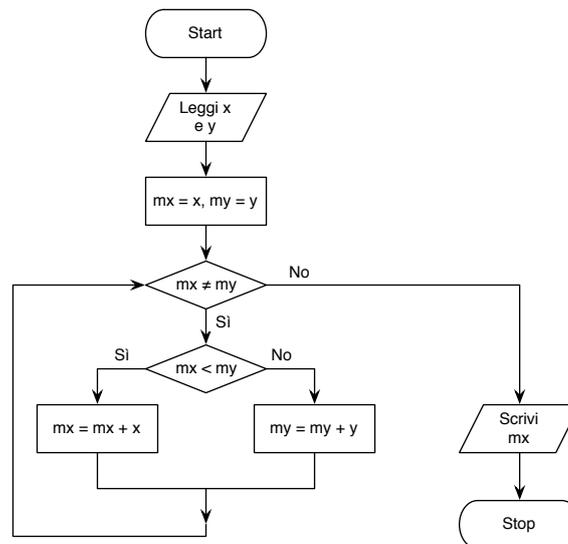


Figura 6: Diagramma di flusso della soluzione dell'Esercizio 6

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: **fin tanto 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'algorithm. 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 }

```

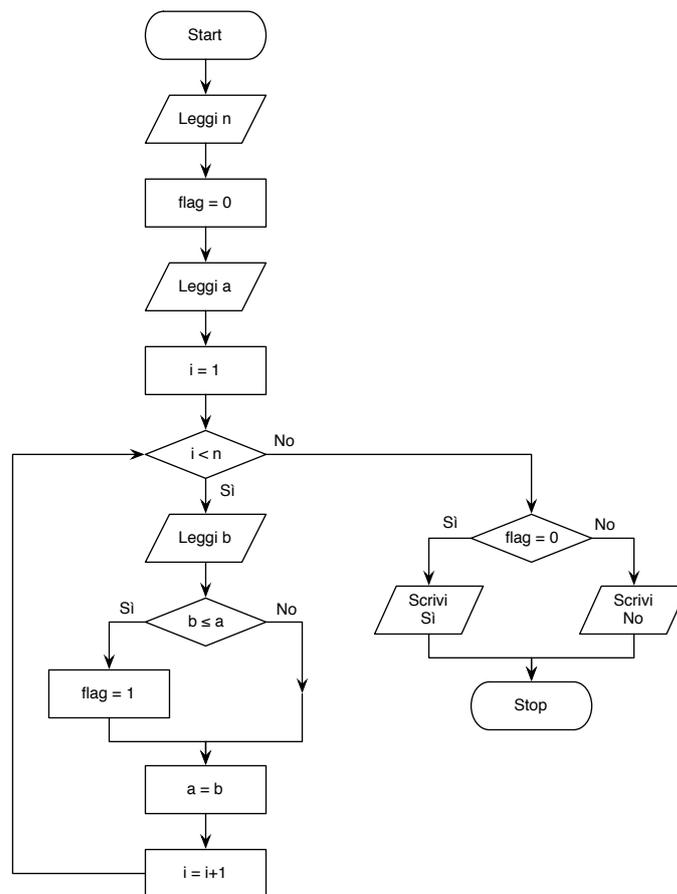


Figura 7: Diagramma di flusso della soluzione dell'Esercizio 7

Riferimenti bibliografici

- [1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data structures and algorithms*, Addison-Wesley, 1987.
- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, seconda edizione, McGraw-Hill, 2005.
- [3] C. Demetrescu, I. Finocchi, G.F. Italiano, *Algoritmi e strutture dati*, McGraw-Hill, 2004.
- [4] M. Liverani, *Programmare in C – Guida al linguaggio attraverso esercizi svolti e commentati*, Società Editrice Esculapio – Progetto Leonardo, 2001.
- [5] M. Liverani, *Qual è il problema? Metodi, strategie risolutive, algoritmi*, Mimesis, 2005.