DAL PROBLEMA ALL'ALGORITMO
AL PROGRAMMA SCRITTO IN C

Scopo principale dell'informatica è risolvere problemi con i calcolatori.

Non tutti i problemi sono risolvibili con i calcolatori. Si può dimostrato che per alcuni problemi non esiste alcun metodo risolutivo automatizzabile.

Alcuni problemi possono non essere risolvibili con i calcolatori perchè hanno infinite soluzioni che richiederebbero un tempo infinito per essere calcolate/stampate. Per altri problemi potrebbe invece non essere stato ancora trovato un metodo di calcolo risolutivo.

Per risolvere un problema con i calcolatori occorre:

- *trovare un procedimento automatizzabile (algoritmo) per risolverlo*
- *implementare l'algoritmo in un linguaggio di programmazione*

Una definizione esatta di algoritmo va oltre lo scopo di questo corso.
Qui ci accontentiamo della seguente definizione.

Un algoritmo è una sequenza di istruzioni perfettamente comprensibili ed eseguibili tali che, se eseguite in un ordine specificato e determinato, permettono la soluzione di un problema in un numero finito di passi.

In questo senso anche una ricetta di cucina molto precisa o le istruzioni per far funzionare un elettrodomestico (se ben scritte e comprensibili) sono esempi di algoritmi.

PROPRIETÀ FONDAMENTALI DI UN ALGORITMO

1 . *istruzioni non ambigue*: ogni operazione prevista dall'algoritmo deve essere univocamente interpretabile dall'esecutore. Questo implica che i risultati ottenuti dall'esecuzione dell'algoritmo non cambiano al variare dell'esecutore (macchina/persona);
2 . *istruzioni eseguibili*: ogni operazione deve essere eseguibile con le risorse a disposizione dell'esecutore;
3. *ordine di esecuzione* d*elle istruzioni deterministico;*
4. *numero finito di passi di esecuzione*: questo implica che il numero totale di istruzioni da eseguire, per ogni insieme di dati di ingresso, è finito e le operazioni da esse specificate devono essere eseguite un numero finito di volte.

PROPRIETÀ AUSPICABILI

1. *corretto;*
2. efficiente: fa uso limitato di risorse (*tempo* di esecuzione e quantità di *memoria* utilizzata);
3. l*eggibile:* facilmente comprensibile;
4. *modificabie*: l'algoritmo deve essere facilmente modificabile, a fronte di (piccole) modifiche nelle specifiche del problema risolto dall'algoritmo.

Problema: calcolare le radici di un'equazione di 2° grado.

Algoritmo:

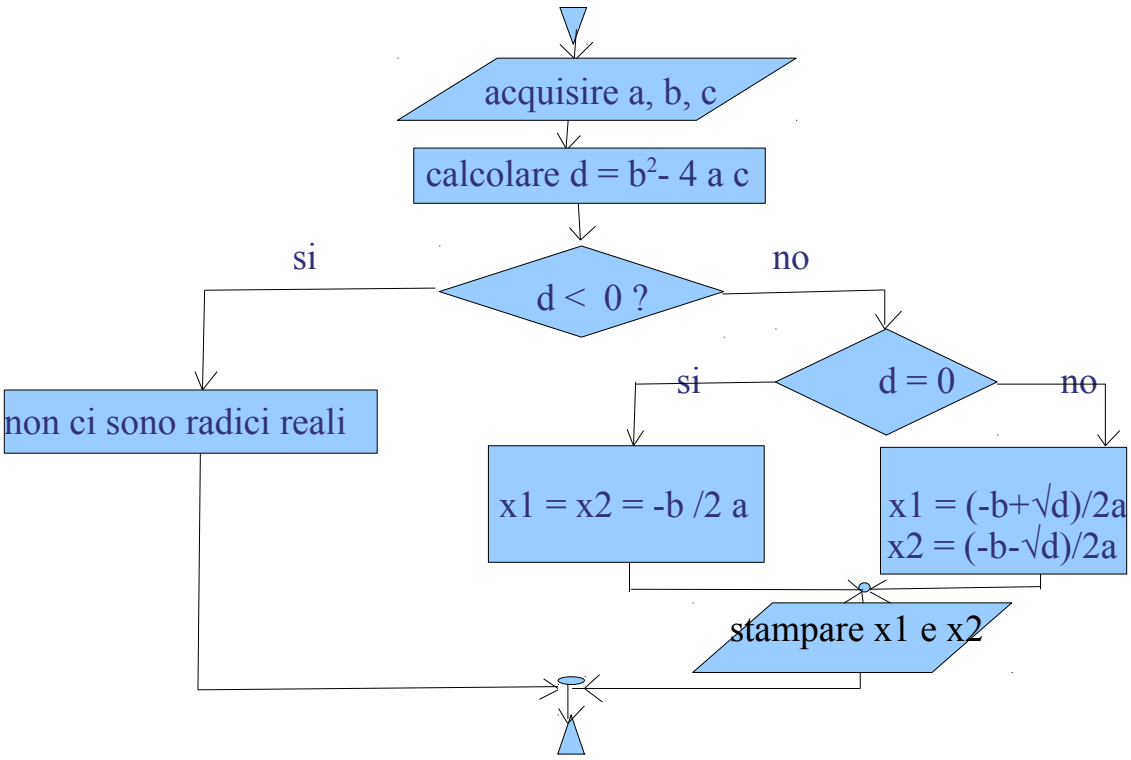
1. Inizio dell'algoritmo.
2. Acquisire i coefficienti a,b,c.
3. Calcolare il valore d = b^2 - 4ac.
4. Se d < 0, allora non esistono radici reali; eseguire l'istruz. 8.
5. Se d = 0, allora esistono due radici reali coincidenti,
   x1 = x2 = -b/2a; eseguire l'istruz. 7.
6. Se d > 0, allora esistono due radici reali distinte,
   x1 = (-b+√d)/2a,
   x2 = (-b-√d)/2a.
7. Comunicare all'esterno i valori x1 e x2.
8. Fine dell'algoritmo.

Le istruzioni eseguibili da un calcolatore moderno possono essere suddivise in 5 categorie:

- *istruzioni di inizio/fine* esecuzione;
- *istruzioni di ingresso(lettura)/uscita(scrittura)*; trasmettono dati tra l'algoritmo ed il mondo esterno;
- *istruzioni operative*; producono un risultato se eseguite;
- i*struzioni di salto*; alterano il normale ordine sequenziale di esecuzione delle istruzioni;
- *istruzioni condizionali*, o di *controllo*; controllano il verificarsi di determinate condizioni ed in base al risultato del controllo determinano quale istruzione va eseguita.

Obiettivo: Rappresentare l'algoritmo in C

Fase intermedia: Rappresentare l'algoritmo con diagrammi di flusso



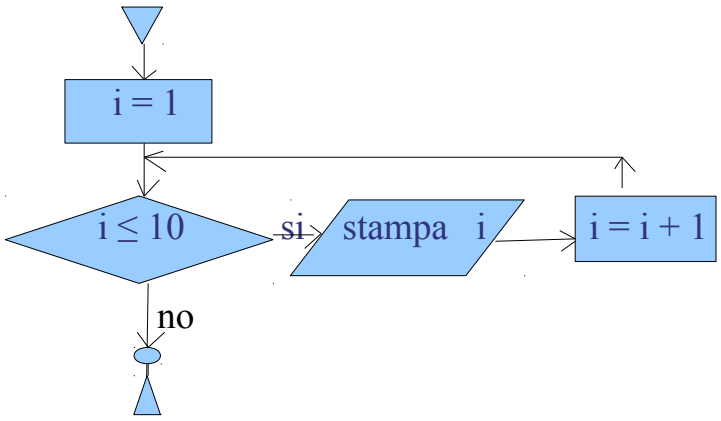
Ecco l'algoritmo scritto in C:

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

int main()
{
    double a,b,c,x1,x2,d;

    printf("Scrivi a, b, c: ");
    scanf("%lf%lf%lf",&a,&b,&c);
    d=b*b-4*a*c;
    if (d<0)
        puts("Non ci sono radici reali");
    else{
        if(d==0){
            x1=-b/(2*a);
            x2=x1;
        } else {
            x1=(-b+sqrt(d))/(2*a);
            x2=(-b-sqrt(d))/(2*a);
        }
        printf("x1= %f\nx2= %f\n",x1,x2);
    }
    return 0;
}
```

Es. Cosa specifica questo diagramma?



Ecco come specificarlo in C in due modi diversi:

```c
#include <stdio.h>

int main()
{
    int i;

    for (i=1;i<=10; i++)
        printf("i=%d\n",i);
    return 0;
}
```

```c
#include <stdio.h>

int main()
{
    int i;

    i=1;
    while (i <=10 ){
        printf("i=%d\n",i);
        i++;
    }
    return 0;
}
```

per risolvere il prolema di stampare gli interi da 1 a 10 su righe distinte.

Diagrammi di flusso
(flow chart / schemi a blocchi)

Formalismo che consente di rappresentare graficamente gli algoritmi.

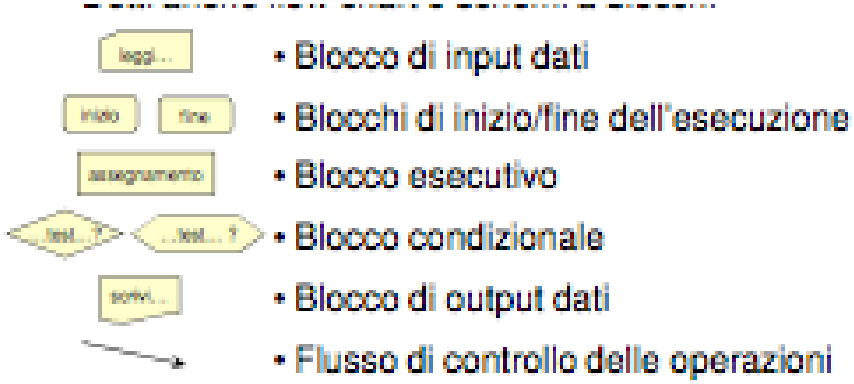
Un diagramma di flusso descrive le istruzioni da eseguire ed il loro ordine di esecuzione.

Le azioni che l'algoritmo deve compiere vengono descritte usando simboli grafici diversi detti **blocchi** (le convenzioni non sono universali), connessi da frecce dette **linee del flusso** di esecuzione delle azioni.

I principali blocchi sono:

- Triangolo o rettangolo spuntato (ist. inizio/fine)
- Rettangolo (ist. operativa)
- Rombo (ist. condizionale)

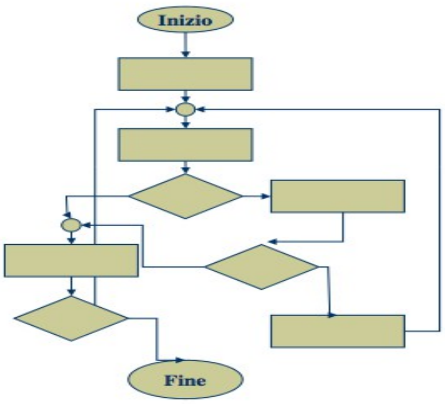
Altri simboli possono essere:



Un diagramma di flusso appare, quindi, come un insieme di blocchi di forme diverse che contengono le istruzioni da eseguire, collegati fra loro da linee orientate che specificano la sequenza in cui i blocchi devono essere eseguiti (flusso del controllo di esecuzione o sequenza di computazione).

Ogni blocco ha uno o più rami in ingresso ma uno solo in uscita a meno che si tratti di un blocco condizionale che ne ha almeno due.

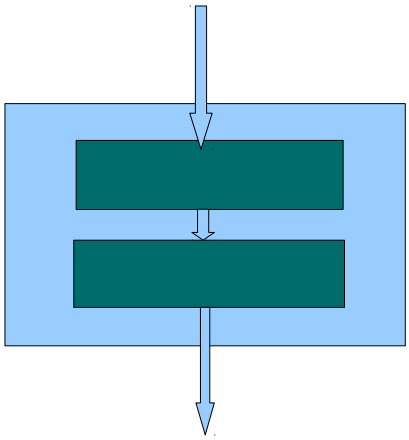
Vi sembra chiaro questo flow-chart?



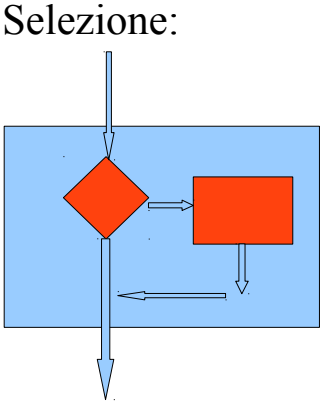
Struttura di controllo: è *combinazione* di blocchi, avente **un solo ramo in entrata ed in uscita** (single entry/single-exit), così il punto di uscita di una struttura di controllo diventa il punto di ingresso di una successiva.

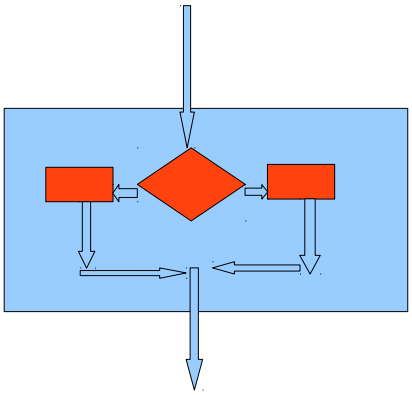
STRUTTURE DI CONTROLLO IN C

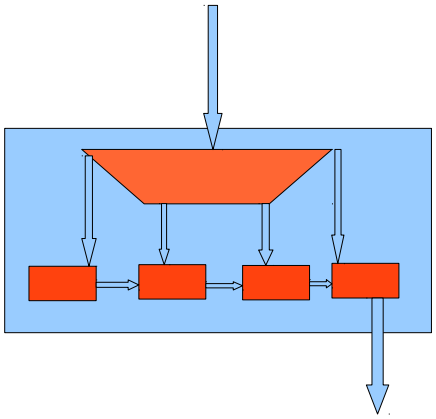
Sequenza:



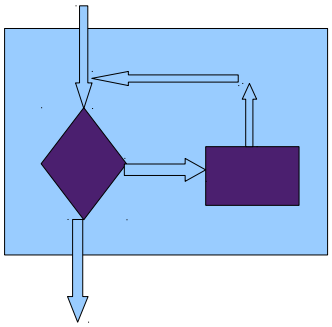
Selezione:

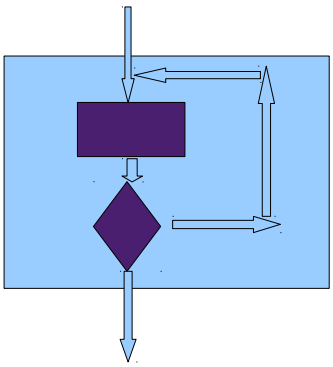


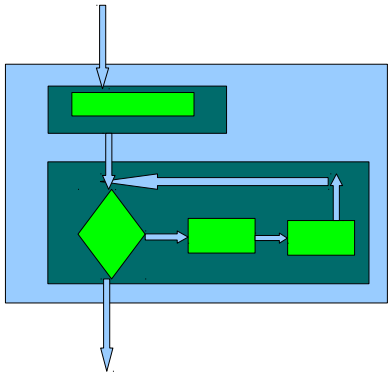




Iterazione:







Il linguaggio C ha molte strutture per il controllo dell'esecuzione:

- **sequenziale** (di default);
- tre di **selezione** (`if, if/else, switch`)
- tre di **iterazione** (`while, do/while, for`)

ESEMPIO DI USO DI IF

```
#include <stdio.h>
int main()
{
     int x, y, max;

     puts("Scrivi due interi:");
     scanf("%d%d", &x,&y);
     max = x;
     if (y > x)
         max = y;
     printf("Il massimo e' %d.\n",max);
     return 0;

}
```

ESEMPIO DI USO DI IF/ELSE

```
#include <stdio.h>
int main()
{
     int x ,y, max;

     puts("Scrivi due interi:");
     scanf("%d%d", &x,&y);
     if (y >= x)
        max = y;
     else
        max = x;
     printf("Il massimo e' %d.\n",max);
     return 0;
}
```

ESEMPIO DI USO DI SWITCH

```
#include <stdio.h>
int main()
{
  int giorno;

  printf("Dammi un intero: ");
  scanf("%d",&giorno);
  printf("Il giorno e'");
  switch(giorno){
     case 1: puts(" lunedi'."); break;
     case 2: puts(" martedi'."); break;
     case 3: puts(" mercoledi'."); break;
     case 4: puts(" giovedi'."); break;
     case 5: puts(" venerdi'."); break;
     case 6: puts(" sabato."); break;
     case 7: puts(" domenica."); break;
     default: puts(" non valido.");
  }
  return 0;
}
```

ESEMPIO DI USO DI WHILE

```
#include <stdio.h>
int main()
{
      int i = 2;

      while (i <= 10){
        printf("i = %d\n",i);
        i = i*2;
      }
      return 0;
}
```

ESEMPIO DI USO DI DO

```
#include <stdio.h>
int main()
{
      int i = 2;

      do {
           printf("i = %d\n",i);
           i *= 2;
      } while(i <= 10);

      return 0;
}
```

ESEMPIO DI USO DI FOR

```
#include <stdio.h>
int main()
{
     int i;

     for (i=2; i <= 10; i*=2)
       printf("i = %d\n",i);

     return 0;
}
```

Regole da seguire per fare programmazione strutturata con il C:

1) Cominciare con la specifica del problema da risolvere, in pratica un rettangolo (azione).
2) Ogni rettangolo (azione) può essere rimpiazzato da due rettangoli (azioni) in sequenza (struttura di controllo **sequenza**).
3) Ogni rettangolo (azione) può essere una struttura di controllo **`if`**, **`if/else`**, **`switch`**, **`while`**, **`do/while`** o **`for`**.
4) Le regole 2 e 3 possono essere applicate in ogni ordine e più volte. Specificano raffinamenti successivi top-down.

PROGRAMMAZIONE STRUTTURATA

La *programmazione strutturata* è una tecnica di programmazione che ha lo scopo di semplificare la struttura di un algoritmo *disciplinando* l'uso delle strutture di controllo utilizzabili all'interno di uno schema blocchi.

In particolare, tale tecnica prevede l'uso di un numero limitato di *strutture di controllo fondamentali*, con un *ingresso* ed una *uscita*:

- **sequenza**
- **selezione**
- **iterazione**

La programmazione strutturata vincola quindi l'utilizzo delle strutture di controllo, ma offre i seguenti vantaggi:

- rende possibile una progettazione di tipo *Top-Down*
- permette la definizione di algoritmi più leggibili, essendo più facile individuare i moduli corrispondenti alle varie parti di cui si compone l'algoritmo
- test, correzione e manutenzione del programma sono perciò più semplici

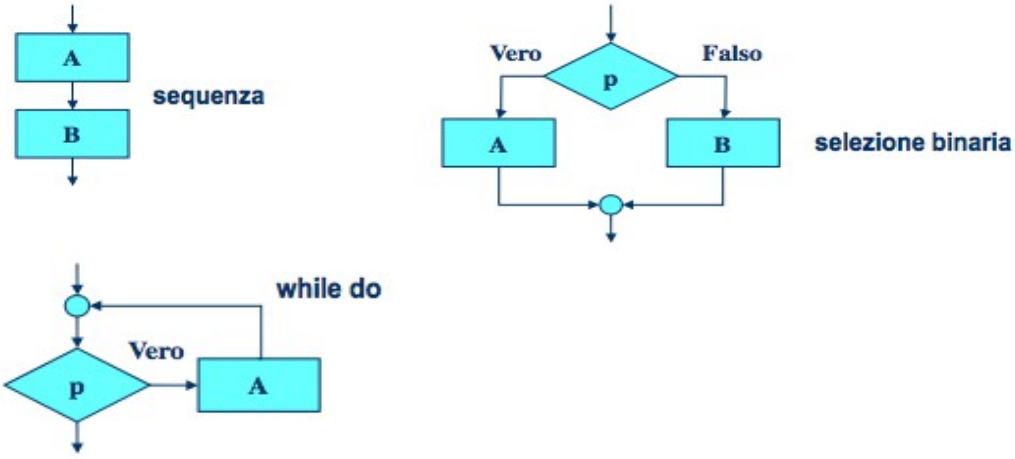
Teorema di Böhm-Jacopini: Tutti gli algoritmi possono essere scritti usando solamente tre strutture di controllo:

- ***Una Sequenziale***
- ***Una di Selezione***
- ***Una di Ripetizione***

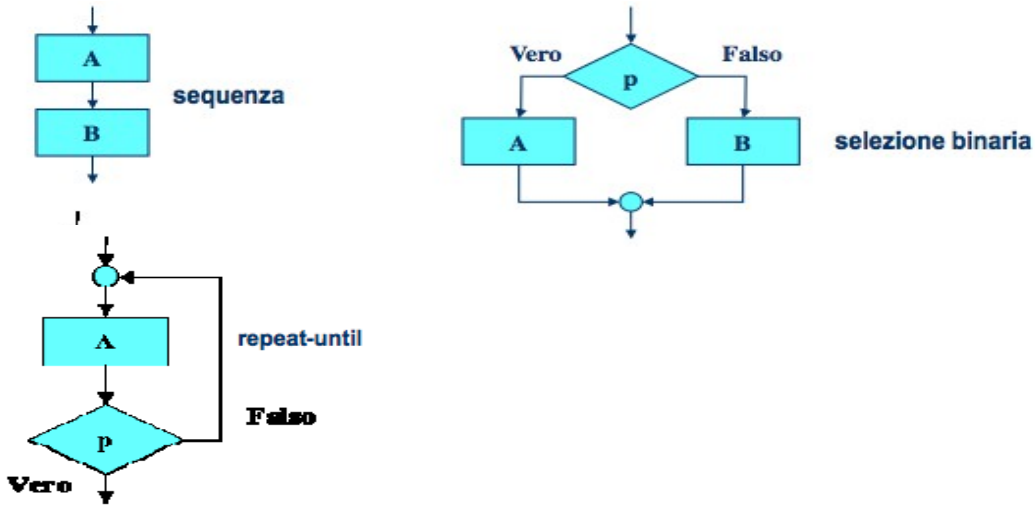
I linguaggi di programmazione tendono però a dotarsi di più strutture di controllo per realizzare programmi più leggibili, più comprensibili e più brevi.

Questo teorema, il cui interesse è soprattutto teorico, ha contribuito alla critica dell'uso sconsiderato delle istruzioni *go to* ed alla definizione delle linee guida della programmazione strutturata, sviluppate negli anni '70.

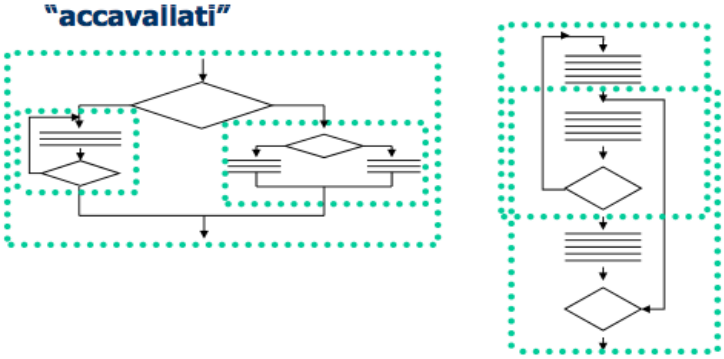
Nella programmazione strutturata potremmo assumere come strutture di controllo fondamentali le tre seguenti:



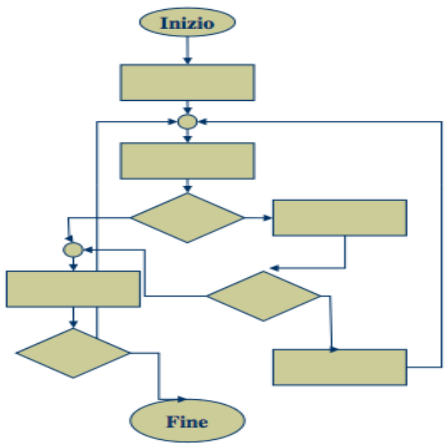
oppure le tre seguenti:



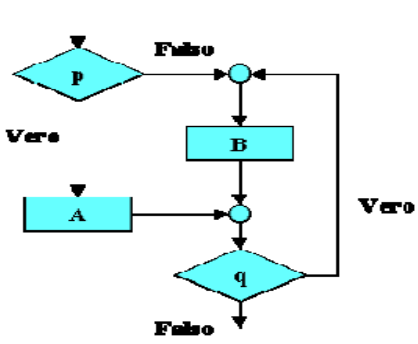
- In un diagramma strutturato non apparirà mai una istruzione di salto incondizionato
- I tre schemi fondamentali possono essere *concatenati,* uno di seguito all'altro, o *nidificati,* uno dentro l'altro; non possono in nessun caso essere "intrecciati" o "accavallati"

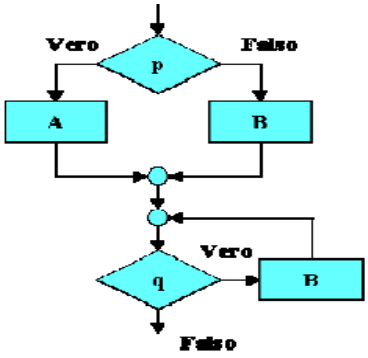


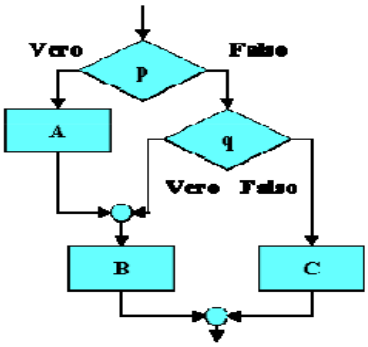
Vi sembra strutturato questo diagramma?

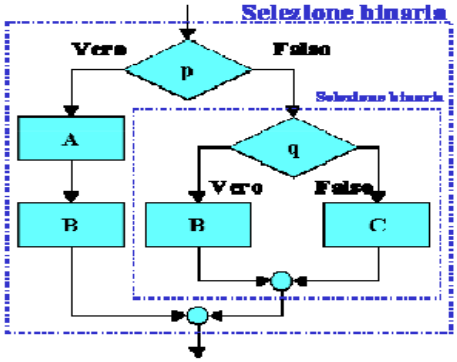


Esempi di passaggio da uno schema a blocchi non strutturato ad uno strutturato:









- **Equivalenza debole**: due flowchart (algoritmi) sono debolmente equivalenti se, per ogni insieme di dati in ingresso, generano gli stessi dati in uscita

- **Equivalenza forte**: due flowchart sono fortemente equivalenti se sono debolmente equivalenti e le rispettive sequenze di computazione sono uguali, per ogni insieme di dati in ingresso

- **Equivalenza fortissima**: due flowchart sono fortissimamente equivalenti se sono fortemente equivalenti ed inoltre in essi compaiono lo stesso numero di volte gli stessi blocchi elementari