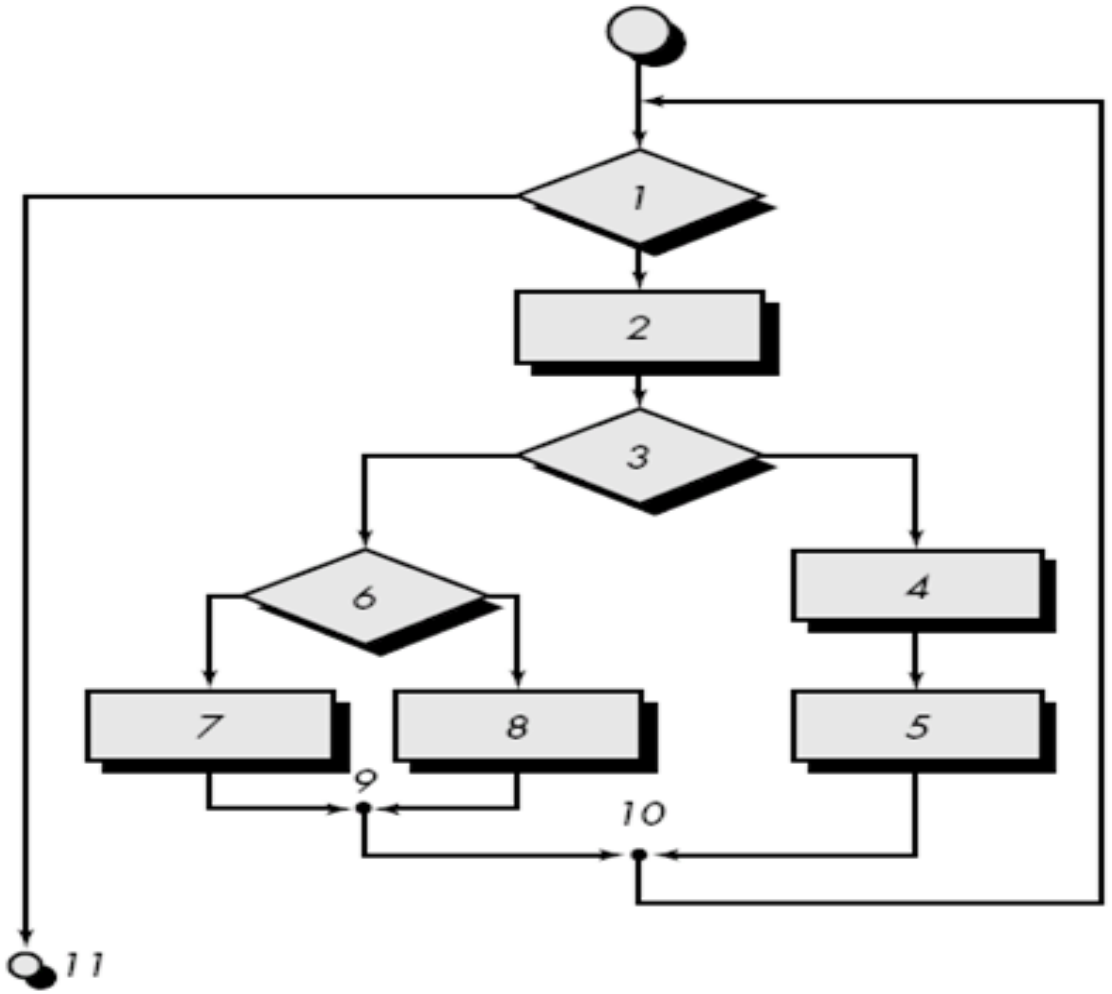
# BASIS PATH TESTING

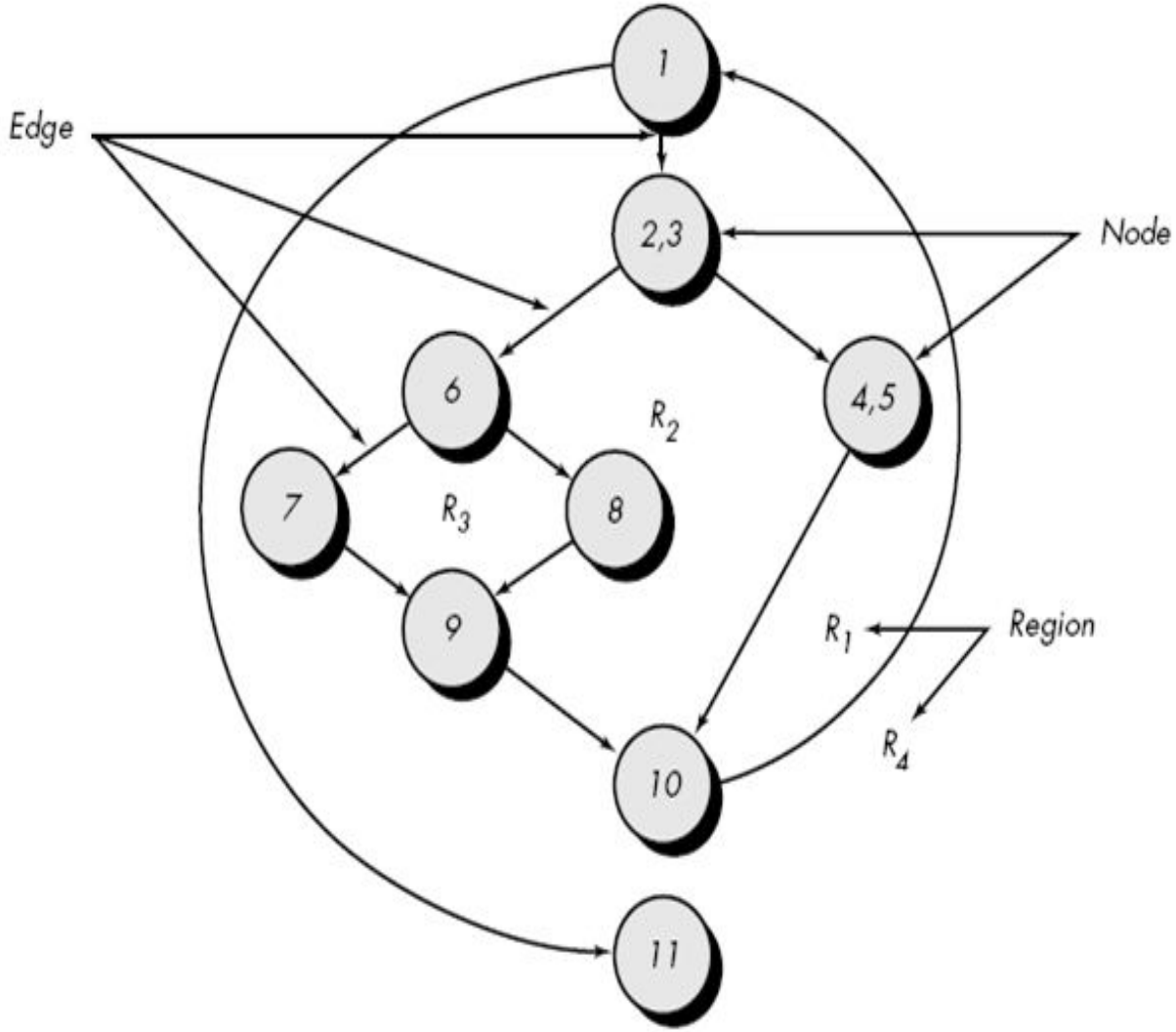
## DERIVING TEST CASES

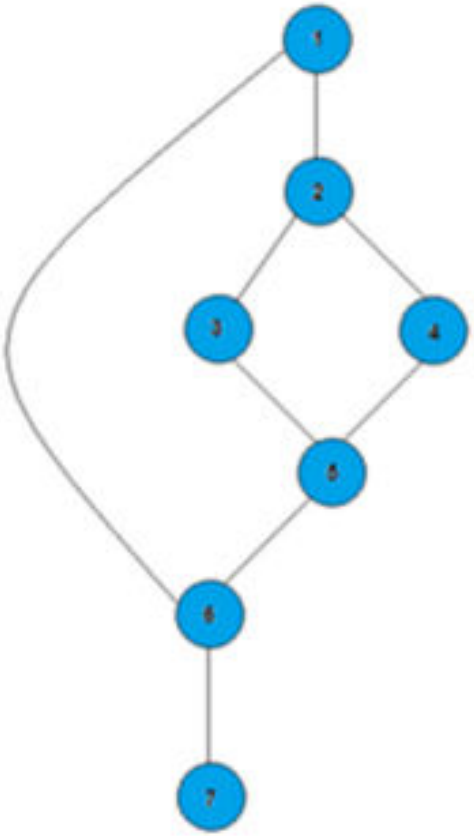
# FlowGraph




1. If A= 50
2. THEN IF B>C
3. THEN A =B
4. ELSE A=C
5. ENDIF
6. ENDIF
7. Print A

# Solution



# Independent program paths

- Path 1: 1-11
- Path 2: 1-2-3-4-5-10-1-11
- Path 3: 1-2-3-6-8-9-10-1-11
- Path 4: 1-2-3-6-7-9-10-1-11
- Path 5: 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

- Path 1: 1,2,3,5,6, 7
- Path 2: 1,2,4,5,6, 7
- Path 3: 1, 6, 7

# Cyclomatic complexity

It is a quantitative measure of independent paths in the source code of a software program.

Cyclomatic complexity can be calculated by using control flow graphs or concerning functions, modules, methods or classes within a software program.

An Independent path is defined as a path that has at least one edge that has not been traversed before in any other path.

1. The *no. of regions* corresponds to the cyclomatic complexity.

2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as

$$V(G) = E - N + 2$$

where E is the number of flow graph edges, N is the number of flow graph nodes.

3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as

$$V(G) = P + 1$$

where P is the number of predicate nodes edges.

- **Regions:**

  V(G) = 4 regions

- **Edge and Node:**

  V(G) = Edge – Node +2

  V(G) = 11 – 9 + 2

  V(G) = 4

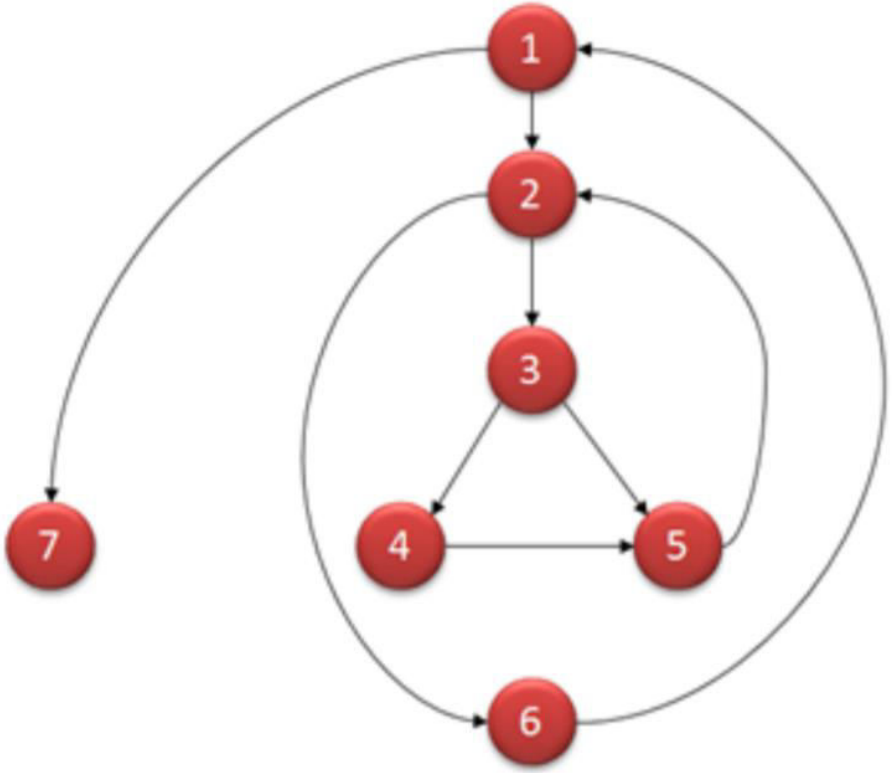
- **Predicate Nodes:**

  V(G) = Predicate Nodes + 1

  V(G) = 3 + 1

  V(G) = 4

# Determine Cyclomatic complexity in all the ways



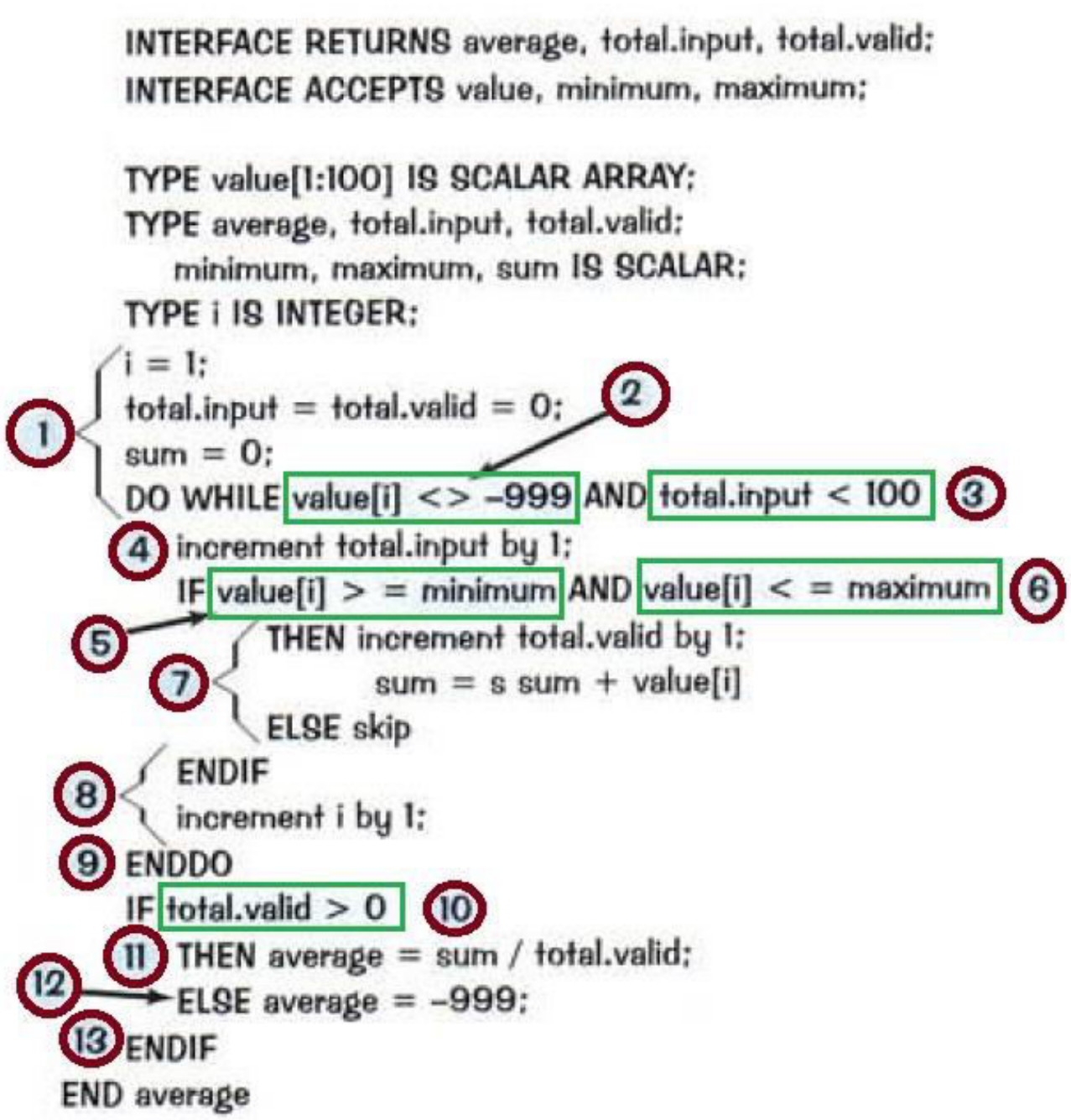
# EXAMPLE

# PROCEDURE

- This procedure computes the average of 100 or fewer numbers that lie between bounding values
- It also computes the sum and the total number valid.

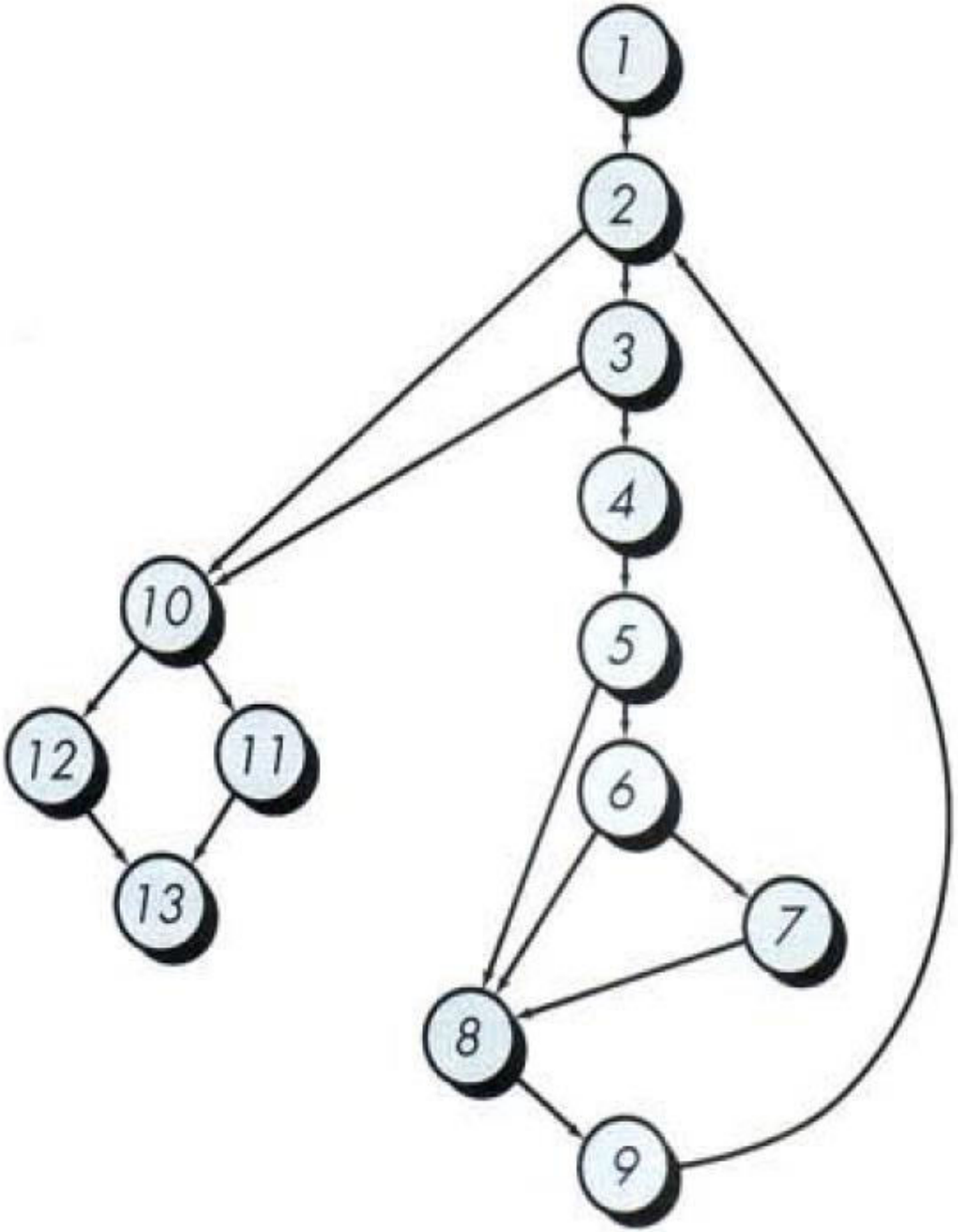
```
    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid:
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100
       increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum
             THEN increment total.valid by 1;
                  sum = s sum + value[i]
             ELSE skip
       ENDIF
       increment i by 1;
    ENDDO
    IF total.valid > 0
       THEN average = sum / total.valid;
       ELSE average = -999;
    ENDIF
END average
```


INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;
TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid:
minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> -999 AND total.input < 100
increment total.input by 1;
IF value[i] > = minimum AND value[i] < = maximum
THEN increment total.valid by 1;
sum = s sum + value[i]
ELSE skip
ENDIF
increment i by 1;
ENDDO
IF total.valid > 0
THEN average = sum / total.valid;
ELSE average = -999;
ENDIF
END average
1
2
3
4
5
6
7
8
9
10
11
12
13

## STEP - 1

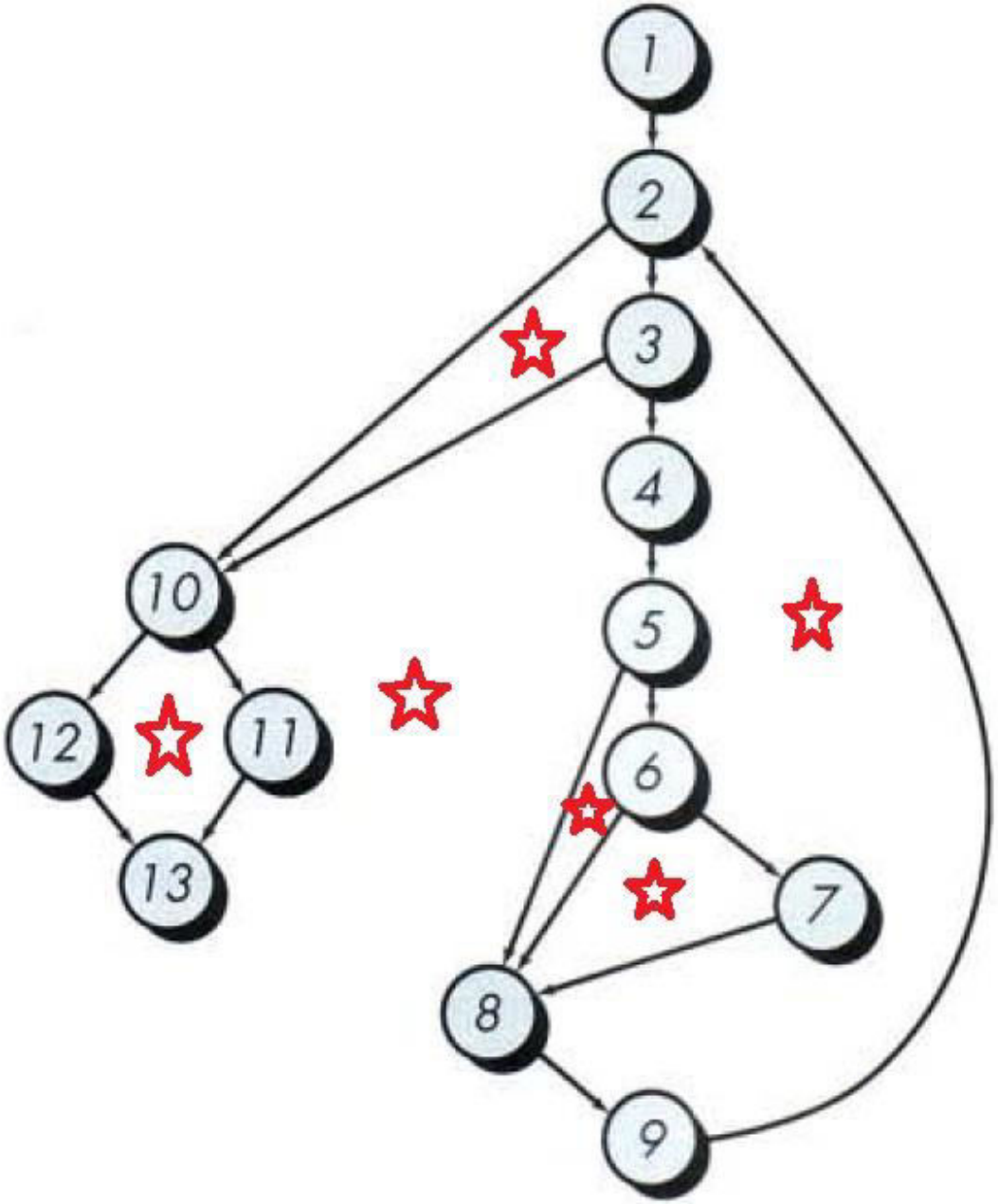
Using the design or code as a foundation, Draw a corresponding flow graph


1
2
3
4
10
5
12
11
6
13
7
8
9

## STEP - 2

Determine the cyclomatic complexity of the resultant flow graph

## REGIONS



- **Regions:**

  V(G) = 6 regions

- **Edge and Node:**

  V(G) = Edge – Node +2

  V(G) = 17 – 13 + 2

  V(G) = 6

- **Predicate Nodes:**

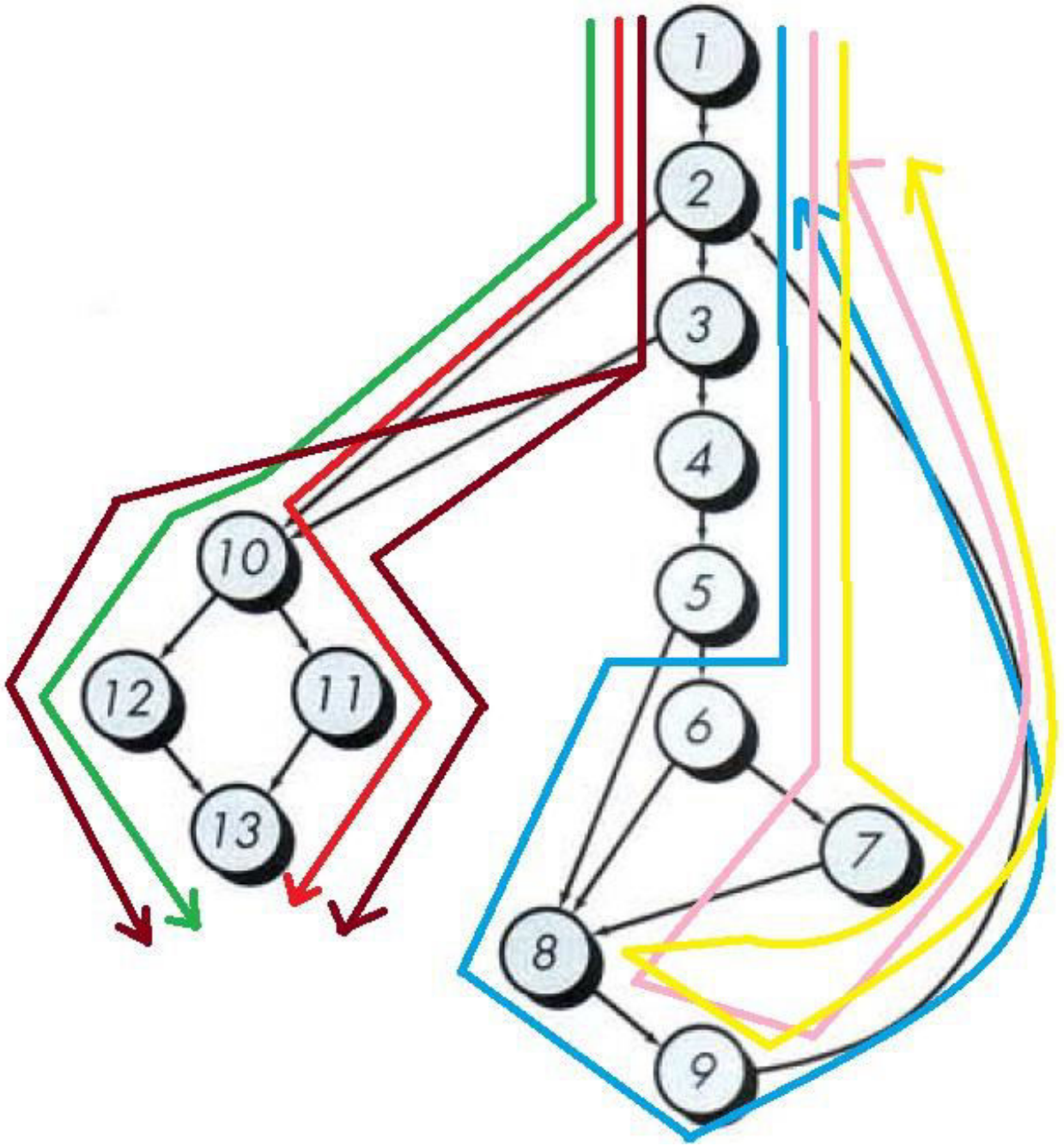
  V(G) = Predicate Nodes + 1

  V(G) = 5 + 1

  V(G) = 6

## STEP - 3

Determine the basis set of linearly independent paths.


1
2
3
4
5
6
7
8
9
10
11
12
13

- PATH – 1 :

  1-2-10-11-13

- PATH – 2 :

  1-2-10-12-13

- PATH – 3 :

  1-2-3-10-11-13

- PATH – 4 :

  1-2-3-4-5-8-9-2-....

- PATH – 5 :

  1-2-3-4-5-6-8-9-2-....

- PATH – 6 :

  1-2-3-4-5-6-7-8-9-2-....

## STEP - 4

Prepare a test cases that will force execution of each path in the basis

- PATH – 1 :

  1-2-10-11-13

- PATH – 2 :

  1-2-10-12-13

- PATH – 3 :

  1-2-3-10-11-13

- PATH – 4 :

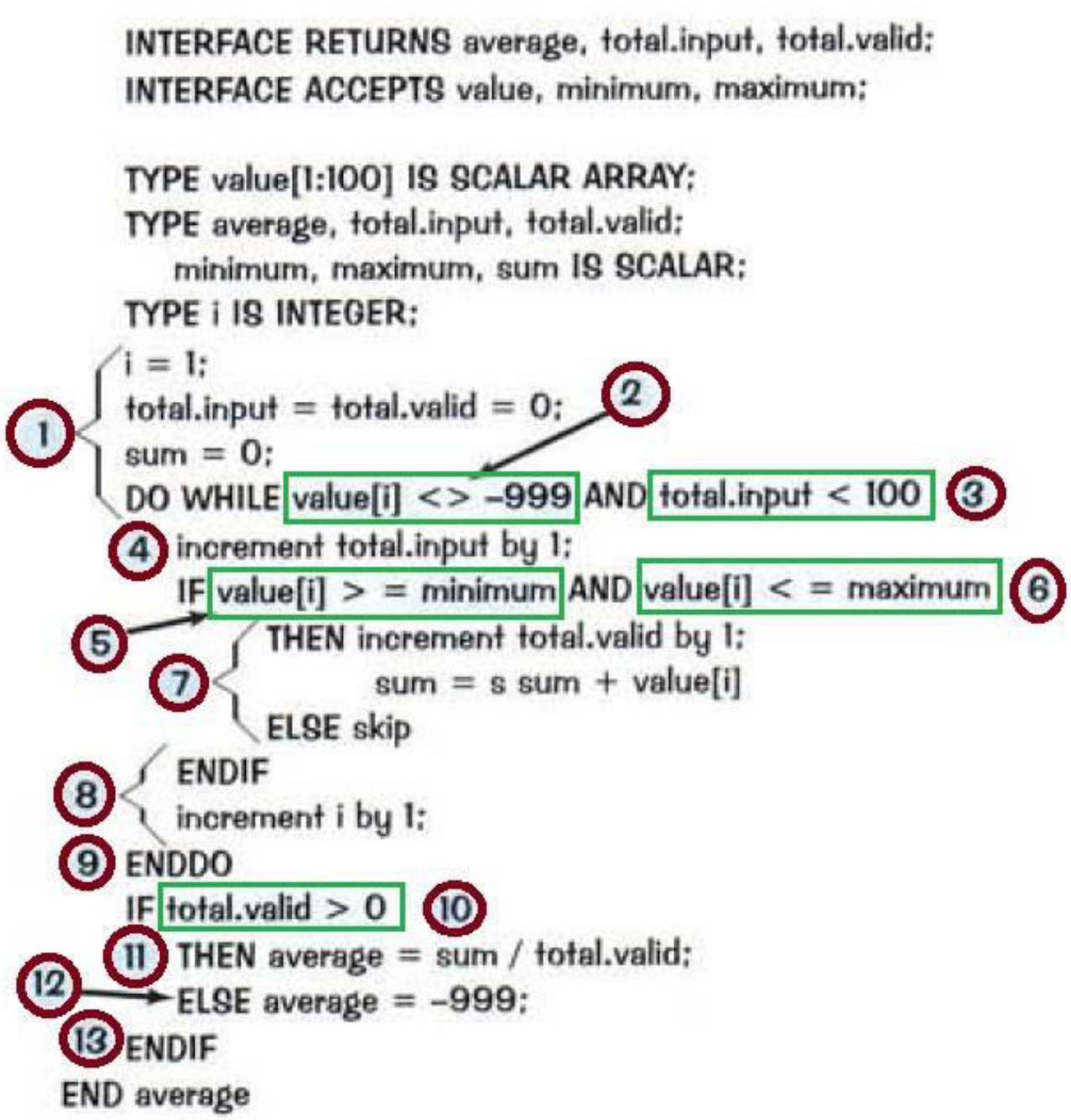
  1-2-3-4-5-8-9-2-....

- PATH – 5 :

  1-2-3-4-5-6-8-9-2-....

- PATH – 6 :

  1-2-3-4-5-6-7-8-9-2-....


INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;
TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid:
minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> -999 AND total.input < 100
increment total.input by 1;
IF value[i] > = minimum AND value[i] < = maximum
THEN increment total.valid by 1;
sum = s sum + value[i]
ELSE skip
ENDIF
increment i by 1;
ENDDO
IF total.valid > 0
THEN average = sum / total.valid;
ELSE average = -999;
ENDIF
END average
1
2
3
4
5
6
7
8
9
10
11
12
13

| INPUT | EXPECTED OUTPUT | PASS / FAIL |
|---|---|---|
| **Test Case – 1<br>Path – 1** | | |
| Sum = 45 , Total valid = 10<br>Value [1] = - 999 | Average = 4.5 | Pass |
| **Test Case – 2<br>Path – 2** | | |
| Sum = 100 , Total valid = 0<br>Value [1] = - 999 | Average = - 999 | Pass |
| **Test Case – 3<br>Path – 3** | | |
| Min = 1, Max = 11,<br>Total.input = 101<br>Value[1, 2, 3, …., 101]<br>Sum = 100 | Average = 0.9 | Pass |

| INPUT | EXPECTED OUTPUT | PASS / FAIL |
|---|---|---|
| **Test Case – 4<br>Path – 4** | | |
| Min = 1, Max = 10,<br>Value[0, 2, 3, …., 10] | Average = - 999 | Pass |
| Min = 1, Max = 10,<br>Value[1, 2, 3, …., 10,0] | Average = 4.5 | Pass |
| **Test Case – 5<br>Path – 5** | | |
| Min = 1, Max = 11,<br>Value[1, 2, 3, …., 11] | Average = 4.5 | Pass |
| Min = 1, Max = 11,<br>Value[11, 2, 3, …., 10] | Average = - 999 | Pass |

| INPUT | EXPECTED OUTPUT | PASS / FAIL |
|---|---|---|
| **Test Case – 6**<br>**Path – 6** | | |
| Min = 1, Max = 11,<br>Value[1, 2, 3, …., 10] | Average = 4.5 | Pass |

# Function to delete element

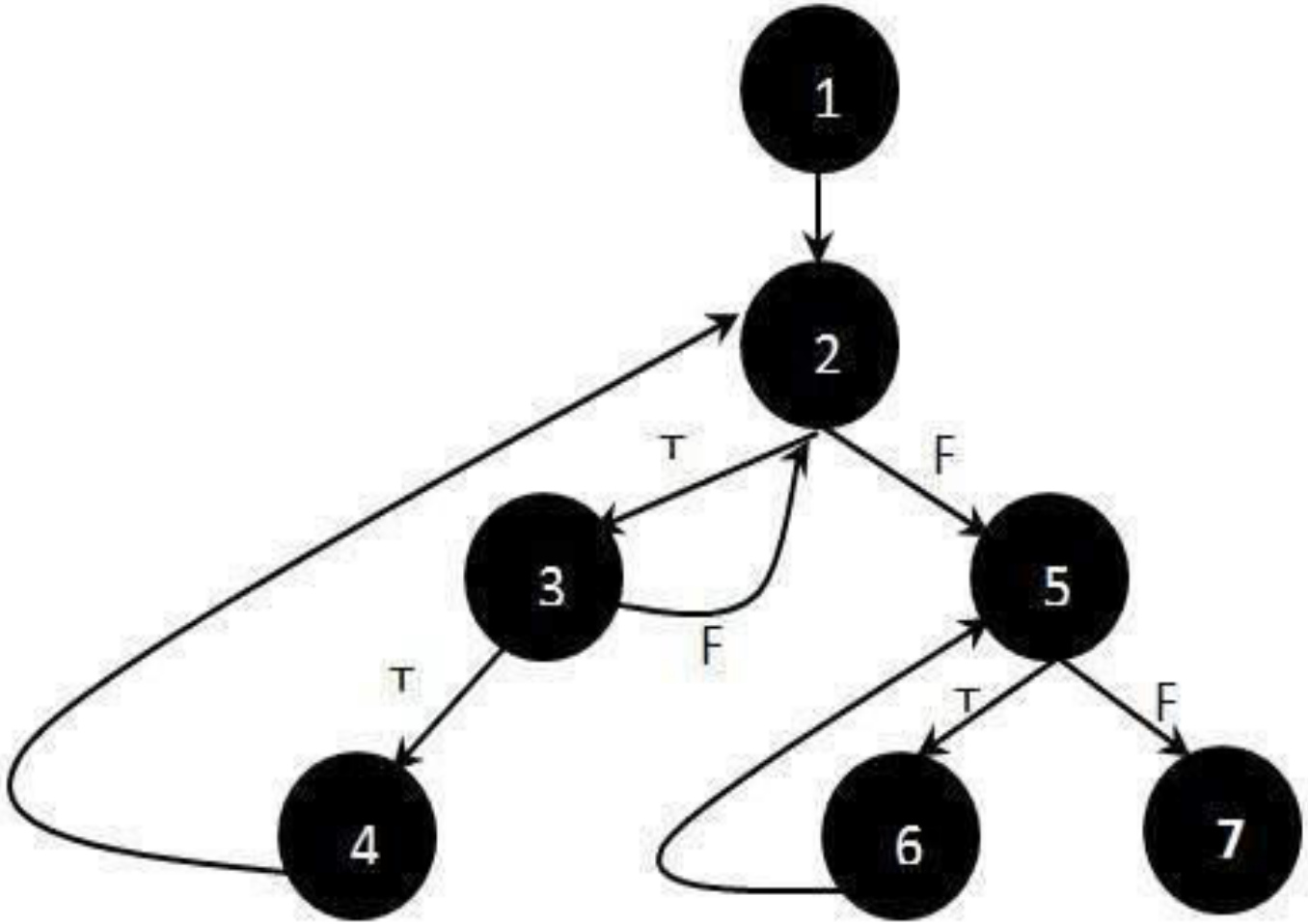
```
Function fn_delete_element (int value, int array_size, int
    array[])
{
    int i;
    location = array_size + 1;

    for i = 1 to array_size
    if ( array[i] == value )
    location = i;
     end if;
     end for;

    for i = location to array_size
    array[i] = array[i+1];
    end for;
    array_size --;
}
```

```
Function fn_delete_element (int value, int
   array_size, int array[])
{
   1 int i;
   location = array_size + 1;
   2 for i = 1 to array_size
   3 if ( array[i] == value )
   4 location = i;
    end if;
    end for;
   5 for i = location to array_size
   6 array[i] = array[i+1];
   end for;
   7 array_size --;
}
```

# Flow graph



# Independent Paths

- Path 1: 1 - 2 - 5 - 7
- Path 2: 1 - 2 - 5 - 6 - 7
- Path 3: 1 - 2 - 3 - 2 - 5 - 6 - 7
- Path 4: 1 - 2 - 3 - 4 - 2 - 5 - 6 - 7

# Advantages of Basic Path Testing

- It helps to reduce the redundant tests
- It focuses attention on program logic
- It helps facilitates analytical versus arbitrary case design
- Test cases which exercise basis set will execute every statement in a program at least once