CSE412
Software Engineering

Nishat Tasnim Niloy

Lecturer

Department of Computer Science and Engineering

Faculty of Science and Engineering

Topic 7

White Box Testing

White Box Testing

- White-box testing, sometimes called glass-box testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.
- Different techniques:
  - Basis path testing
  - Control structure testing

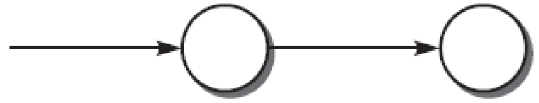
Path Testing

- Path testing is based on control structure of the program for which flow graph is prepared.
- Path testing requires complete knowledge of the program's structure.
- Path testing is closer to the developer and used by him to test his module.
- The effectiveness of path testing gets reduced with the increase in size of software under test.
- Choose enough paths in a program such that maximum logic coverage is achieved.

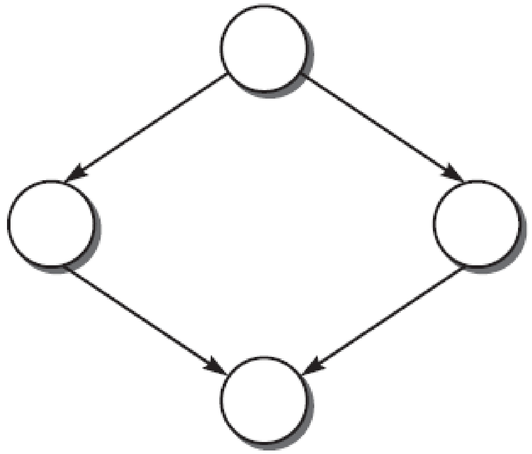
Control Flow Graph

- The control flow graph is a graphical representation of control structure of a program.
- Some Term:
    - **Node:** It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
    - **Edges or links:** They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
    - **Decision node:** A node with more than one arrow leaving it is called a decision node.
    - **Junction node:** A node with more than one arrow entering it is called a junction.
    - **Regions:** Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

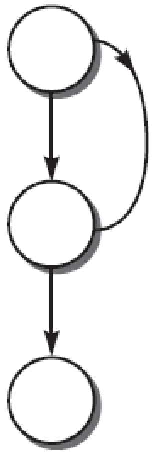
Flow Graph Notations



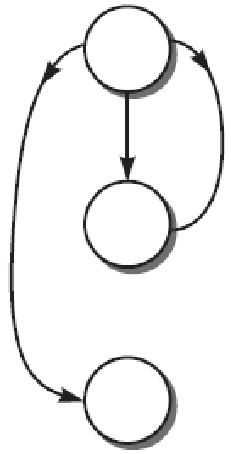
(a) Sequence



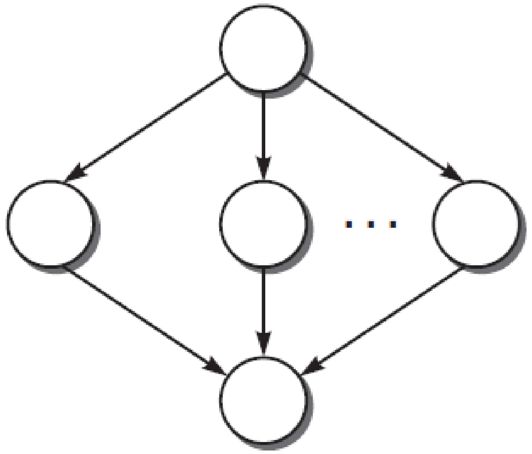
(b) If-Then-Else



(c) Do-While



(d) While-Do
/For Loop



(e) Switch-Case

PATH TESTING TERMINOLOGY

- Path
  - A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.
- Segment
  - Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction process-decision, decision-process-junction, decision-process-decision).
- Length of a Path
  - The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed.
- Independent Path
  - An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions.

Cyclomatic Complexity (CC)

- Cyclomatic complexity is a software metric used to measure the complexity of a program.
- The idea is to measure the complexity by considering the number of paths in the control graph of the program.
- But even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore, we will consider only independent paths.
- Cyclomatic complexity number can be derived through any of the following three formulae:

$$V(G) = e - n + 2p$$

where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.

DD Graph (Decision-to-Decision Graph)

- For a DD graph, the following actions must be done:
  - Put the line numbers on the execution statements of the program. Start numbering the statements after declaring the variables, if no variables have been initialized. Otherwise, start from the statement where a variable has been initialized.
  - Put the sequential statements in one node. For example, statements 1, 2, and 3 have been put inside one node.
  - Put the edges between the nodes according to their flow of execution.
  - Put alphabetical numbering on each node like A, B, etc.

Example 1

- Draw the DD graph for the program.
- Calculate the cyclomatic complexity of the program using all the methods.
- List all independent paths.
- Design test cases from independent paths.

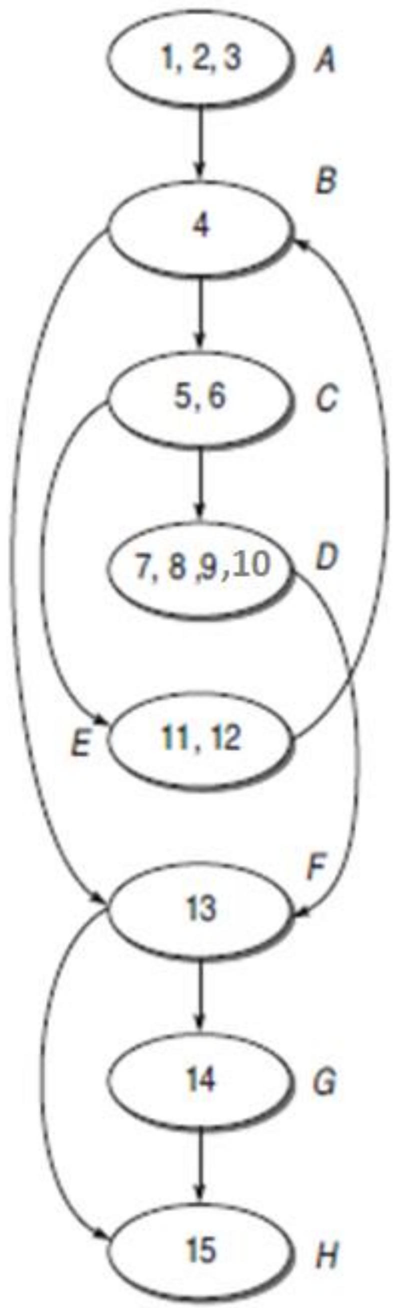
- Consider the following program segment:

```
main()
  {
   int number, index;
1. printf("Enter a number");
2. scanf("%d, &number);
3. index = 2;
4. while(index <= number – 1)
5. {
6.    if (number % index == 0)
7.    {
8.        printf("Not a prime number");
9.        break;
10.   }
11.    index++;
12. }
13. if(index == number)
14.       printf("Prime number");
15.} //end main
```

DD Graph

- The DD graph of the program is shown in the next Figure:

```
main()
  {
    int number, index;
1.  printf("Enter a number");
2.  scanf("%d, &number);
3.  index = 2;
4.  while(index <= number - 1)
5.  {
6.     if (number % index == 0)
7.     {
8.         printf("Not a prime number");
9.         break;
10.    }
11.    index++;
12. }
13. if(index == number)
14.        printf("Prime number");
15.} //end main
```



Calculating CC

Cyclomatic complexity of main()

V(G) = e – n + 2 * p

= 10 – 8 + 2

= 4

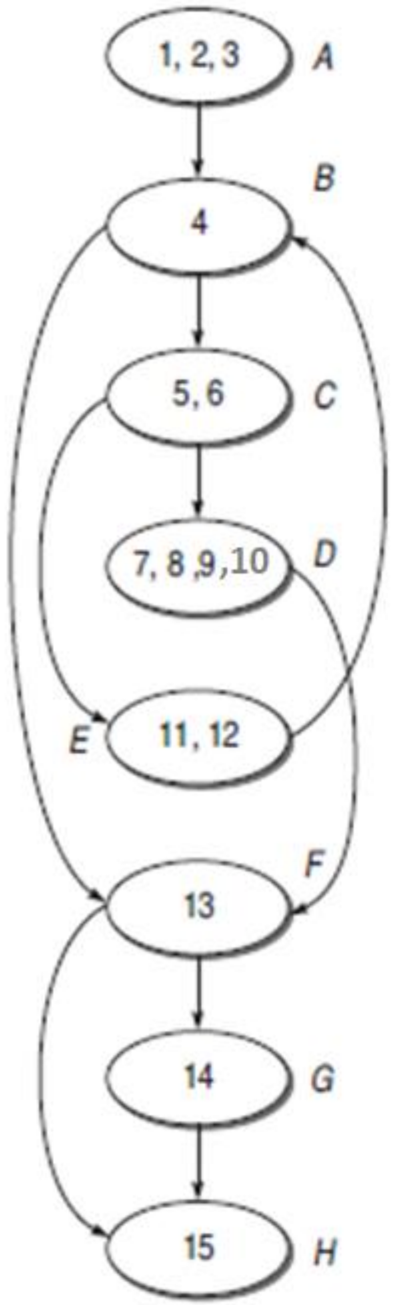
Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) A-B-F-H

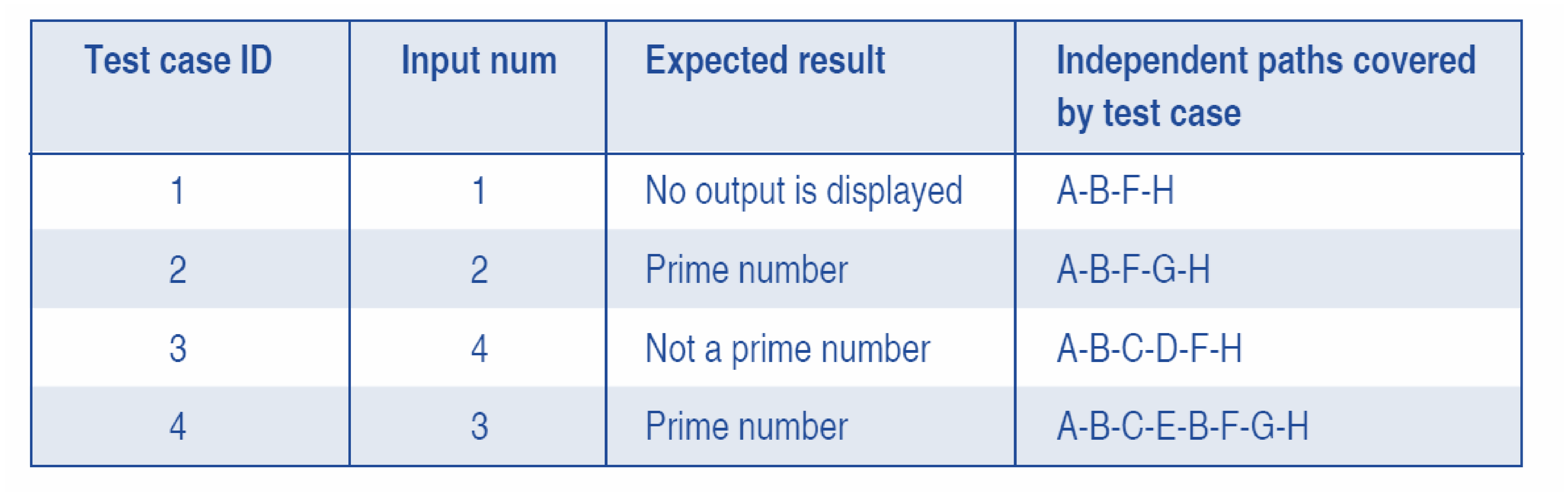
(ii) A-B-F-G-H

(iii) A-B-C-E-B-F-G-H

(iv) A-B-C-D-F-H



Test Case Design from the Independent Paths



Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Example 2

- Draw the DD graph for the program.
- Calculate the cyclomatic complexity of the program using all the methods.
- List all independent paths.
- Design test cases from independent paths.

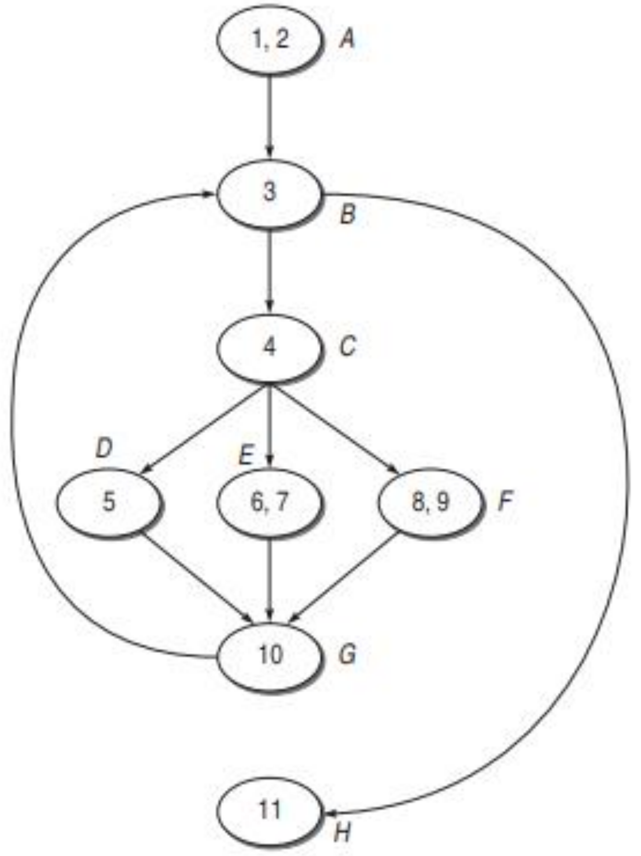
- Consider the following program segment:

```
   main()
       {
         char string [80];
         int index;
1.       printf(“Enter the string for checking its characters”);
2.       scanf(“%s”, string);
3.       for(index = 0; string[index] != ‘\0’; ++index) {
4.          if((string[index] >= ‘0’) && (string[index] <=‘9’))
5.             printf(“%c is a digit”, string[index]);
6.          else if ((string[index] >= ‘A’) && (string[index]
           <‘Z’))||((string[index] >= ‘a’) && (string[index] <‘z’)))
7.             printf(“%c is an alphabet”, string[index]);
8.          else
9.             printf(“%c is a special character”, string[index]);
10.       }
11. }
```

DD Graph

The DD graph of the program is shown in the next Figure:

```
    main()
        {
          char string [80];
          int index;
 1.       printf(“Enter the string for checking its characters”);
 2.       scanf(“%s”, string);
 3.       for(index = 0; string[index] != ‘\0’; ++index) {
 4.          if((string[index] >= ‘0’) && (string[index] <=‘9’))
 5.             printf(“%c is a digit”, string[index]);
 6.          else if ((string[index] >= ‘A’) && (string[index]
            <‘Z’))||((string[index] >= ‘a’) && (string[index] <‘z’)
 7.             printf(“%c is an alphabet”, string[index]);
 8.          else
 9.             printf(“%c is a special character”, string[index]);
 10.      }
 11. }
```



Calculating CC

Cyclomatic complexity of main()

V(G) = e – n + 2 * p

= 10 – 8 + 2

= 4

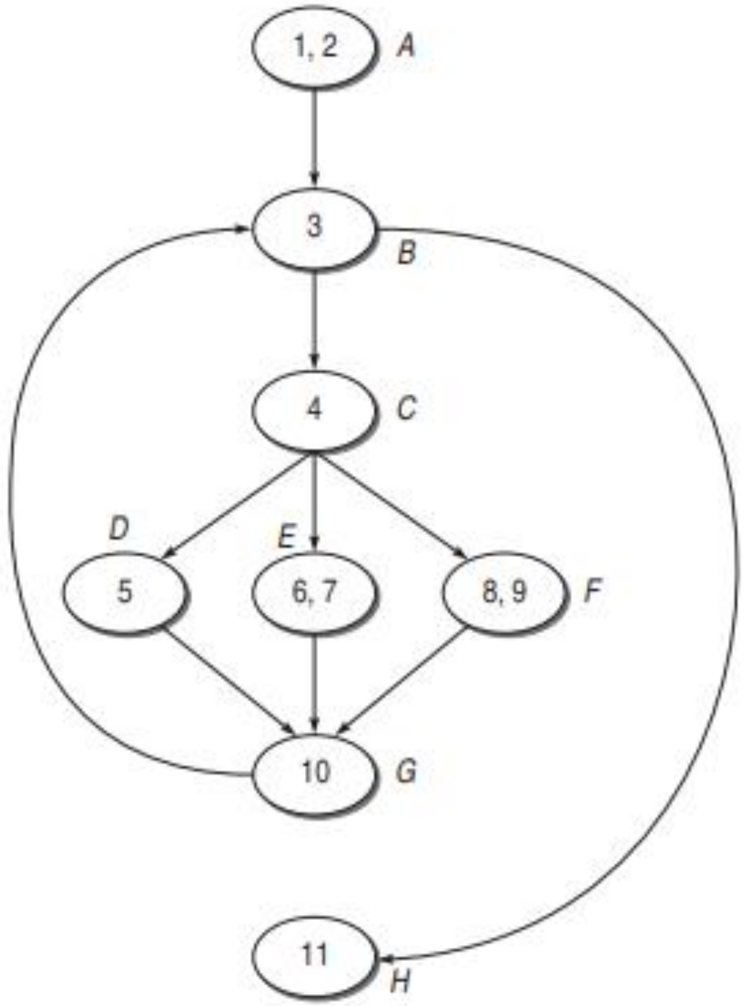
Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) A-B-H

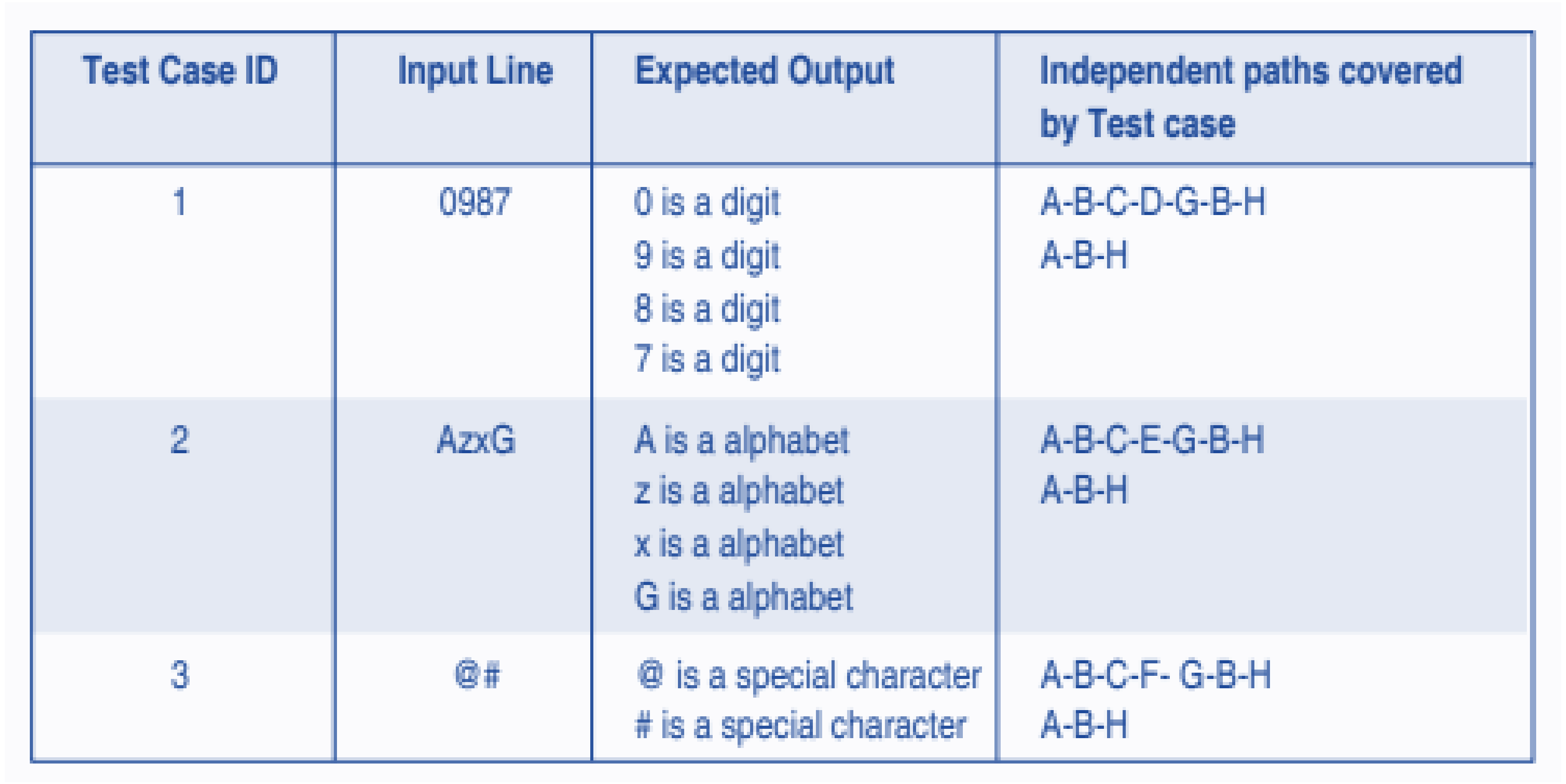
(ii) A-B-C-D-G-B-H

(iii) A-B-C-E-G-B-H

(iv) A-B-C-F-G-B-H



Test Case Design from the Independent Paths



Test Case ID	Input Line	Expected Output	Independent paths covered by Test case
1	0987	0 is a digit 9 is a digit 8 is a digit 7 is a digit	A-B-C-D-G-B-H A-B-H
2	AzxG	A is a alphabet z is a alphabet x is a alphabet G is a alphabet	A-B-C-E-G-B-H A-B-H
3	@#	@ is a special character # is a special character	A-B-C-F- G-B-H A-B-H

Example 3

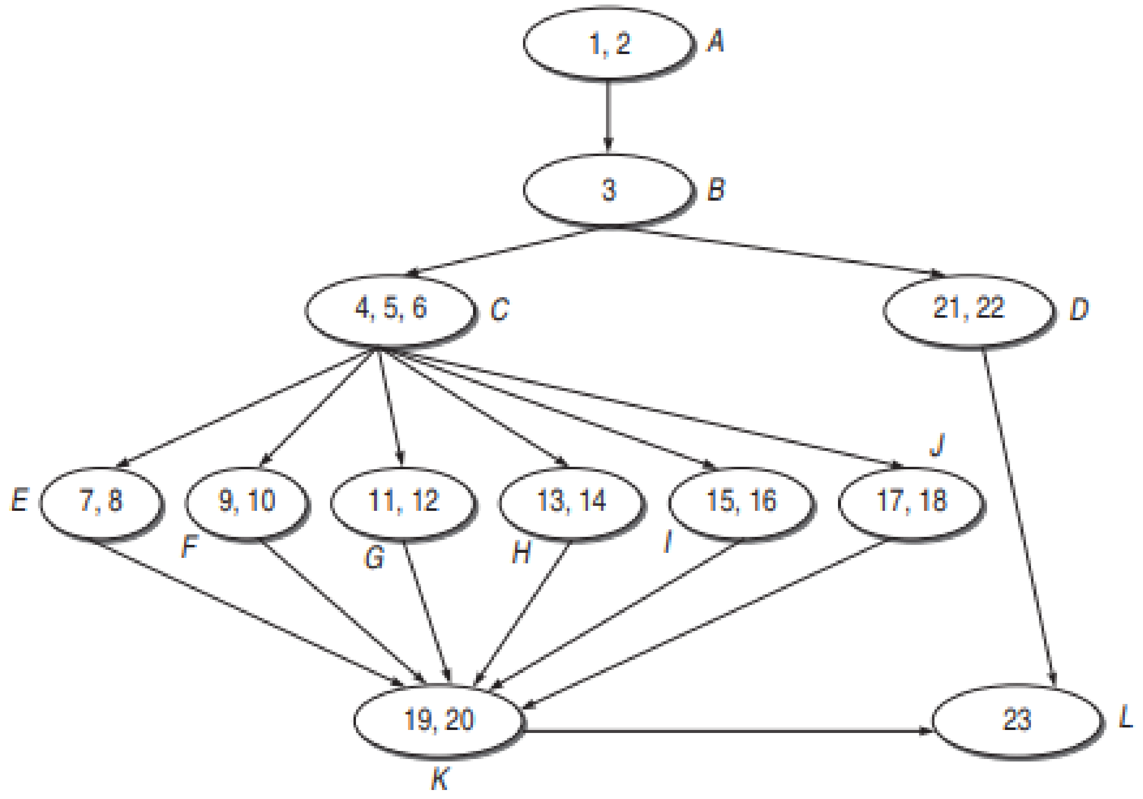
- Draw the DD graph for the program.
- Calculate the cyclomatic complexity of the program using all the methods.
- List all independent paths.
- Design test cases from independent paths.

```
main(){
          char chr;
1.        printf (“Enter the special character\n”);
2.        scanf (%c”, &chr);
3.        if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) && (chr !=
                    52) && (chr != 53) && (chr != 54) && (chr != 55) && (chr != 56)
                    && (chr != 57)
4.        {
5.              switch(chr)
6.                  {
7.                  Case ‘*’: printf(“It is a special character”);
8.                  break;
9.                  Case ‘#’: printf(“It is a special character”);
10.                 break;
11.                 Case ‘@’: printf(“It is a special character”);
12.                 break;
13.                 Case ‘!’: printf(“It is a special character”);
14.                 break;
15.                 Case ‘%’: printf(“It is a special character”);
16.                 break;
17.                 default : printf(“You have not entered a special character”);
18.                 break;
19.                 }// end of switch
20.           } // end of If
21.       else
22.           printf(“You have not entered a character”);
23.   } // end of main()
```

DD Graph

The DD graph of the program is shown in the next Figure:

```
main(){
           char chr;
1.         printf (“Enter the special character\n”);
2.         scanf (%c”, &chr);
3.         if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) && (chr !=
                        52) && (chr != 53) && (chr != 54) && (chr != 55) && (chr != 56)
                        && (chr != 57)
4.         {
5.              switch(chr)
6.                 {
7.                      Case ‘*’: printf(“It is a special character”);
8.                      break;
9.                      Case ‘#’: printf(“It is a special character”);
10.                     break;
11.                     Case ‘@’: printf(“It is a special character”);
12.                     break;
13.                     Case ‘!’: printf(“It is a special character”);
14.                     break;
15.                     Case ‘%’: printf(“It is a special character”);
16.                     break;
17.                     default : printf(“You have not entered a special character”);
18.                     break;
19.                 }// end of switch
20.        } // end of If
21.        else
22.           printf(“You have not entered a character”);
23.    } // end of main()
```



Calculating CC

Cyclomatic complexity of main()

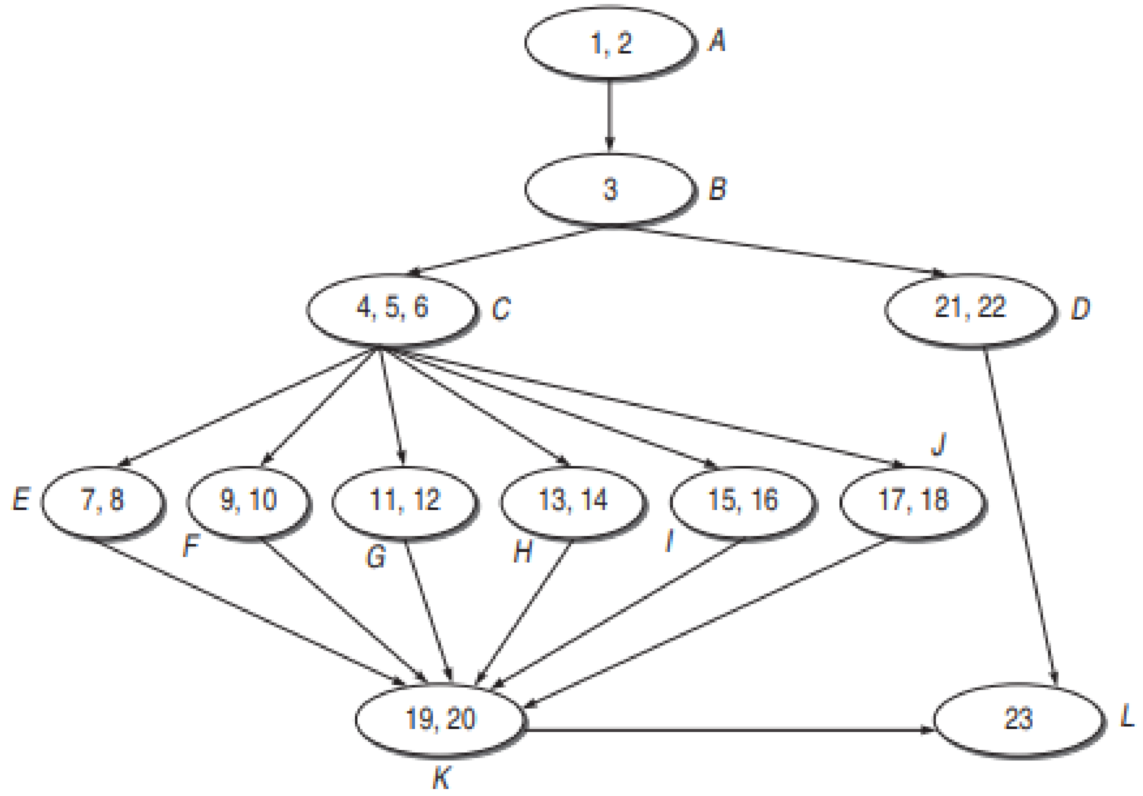
V(G) = e – n + 2p

= 17 – 12 + 2

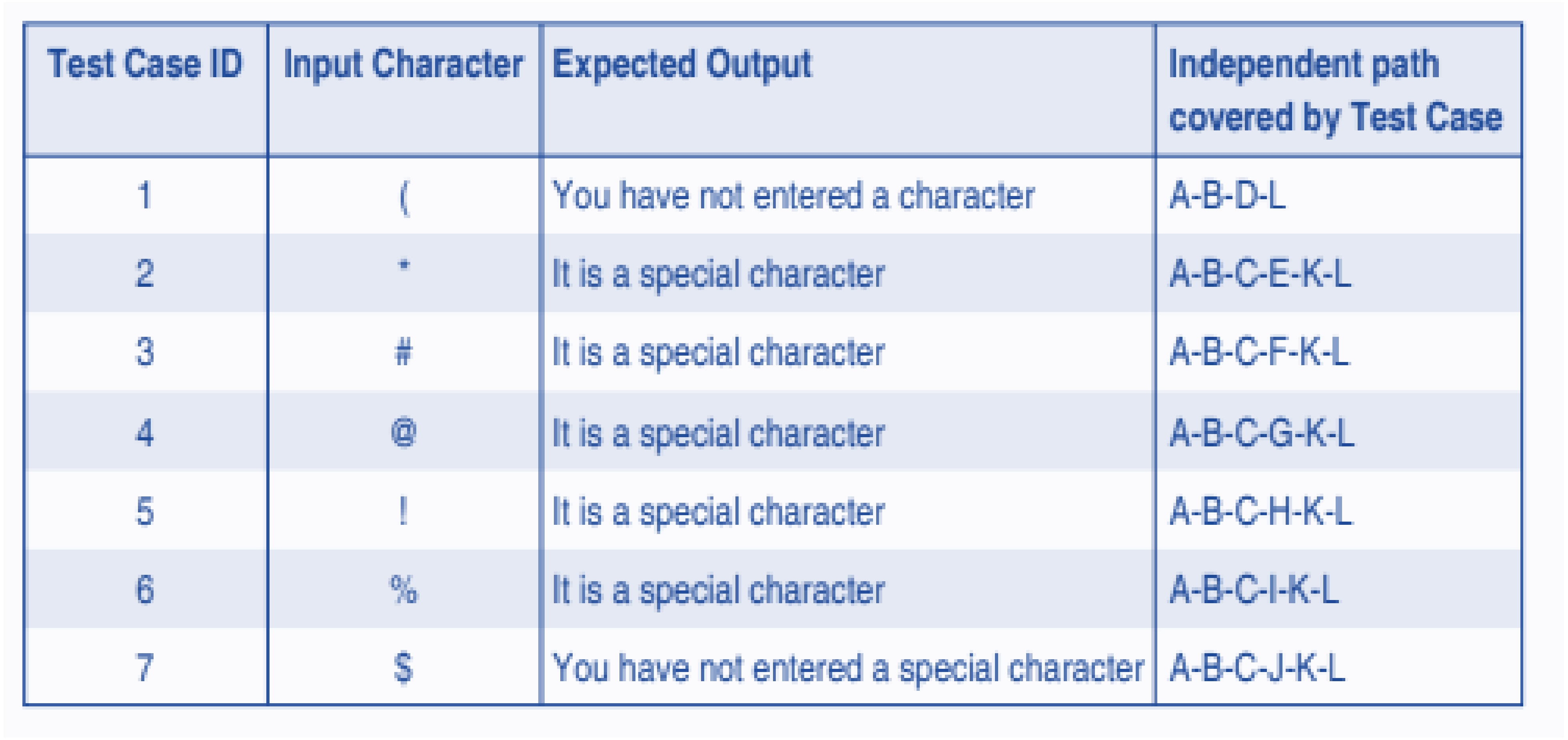
= 7

Since the cyclomatic complexity of the graph is 7, there will be 7 independent paths in the graph as shown below:

1. A-B-D-L
2. A-B-C-E-K-L
3. A-B-C-F-K-L
4. A-B-C-G-K-L
5. A-B-C-H-K-L
6. A-B-C-I-K-L
7. A-B-C-J-K-L



Test Case Design from the Independent Paths



Test Case ID	Input Character	Expected Output	Independent path covered by Test Case
1	(	You have not entered a character	A-B-D-L
2	*	It is a special character	A-B-C-E-K-L
3	#	It is a special character	A-B-C-F-K-L
4	@	It is a special character	A-B-C-G-K-L
5	!	It is a special character	A-B-C-H-K-L
6	%	It is a special character	A-B-C-I-K-L
7	$	You have not entered a special character	A-B-C-J-K-L

Example 4

- Draw the DD graph for the program.
- Calculate the cyclomatic complexity of the program using all the methods.
- List all independent paths.
- Design test cases from independent paths.

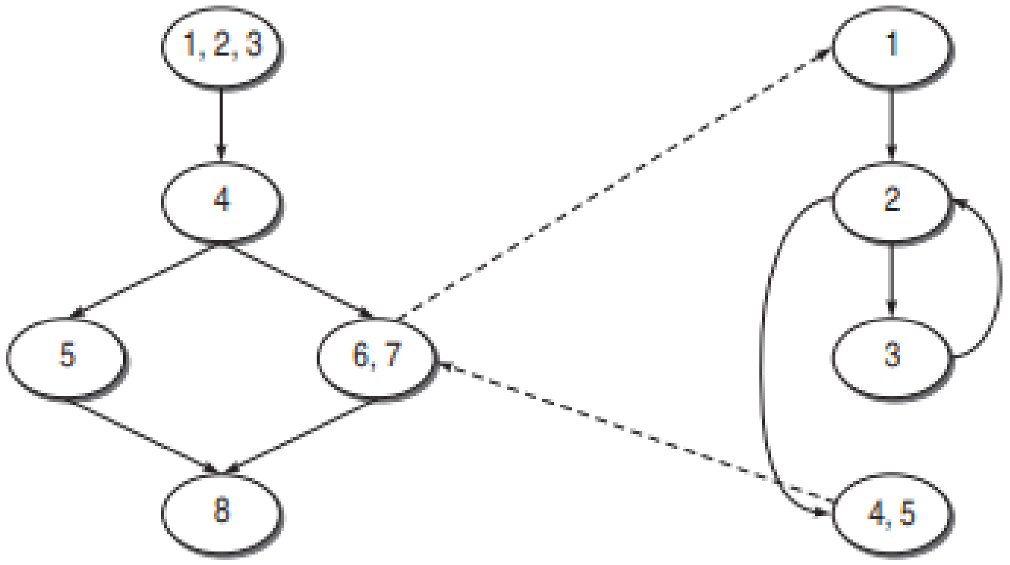
- Consider the following program segment:

```
main()
    {
      int number;
      int fact();
1.    clrscr();
2.    printf("Enter the number whose factorial is to be found out");
3.    scanf("%d", &number);
4.    if(number <0)
5.       printf("Factorial cannot be defined for this number);
6.    else
7.       printf("Factorial is %d", fact(number));
8.  }

int fact(int number)
   {
     int index;
1.   int product =1;
2.   for(index=1; index<=number; index++)
3.      product = product * index;
4.   return(product);
5. }
```

DD Graph

The DD graph of the program is shown in the next Figure:

```
     main()
          {
            int number;
            int fact();
 1.         clrscr();
 2.         printf("Enter the number whose factorial is to be found out");
 3.         scanf("%d", &number);
4.          if(number <0)
5.              printf("Factorial cannot be defined for this number);
6.          else
7.              printf("Factorial is %d", fact(number));
8.        }

     int fact(int number)
        {
            int index;
1.          int product =1;
2.          for(index=1; index<=number; index++)
3.              product = product * index;
4.          return(product);
5.  }
```



(a) Flow graph for main ()   (b) Flow graph for fact ()

Calculating CC

Cyclomatic complexity of main()

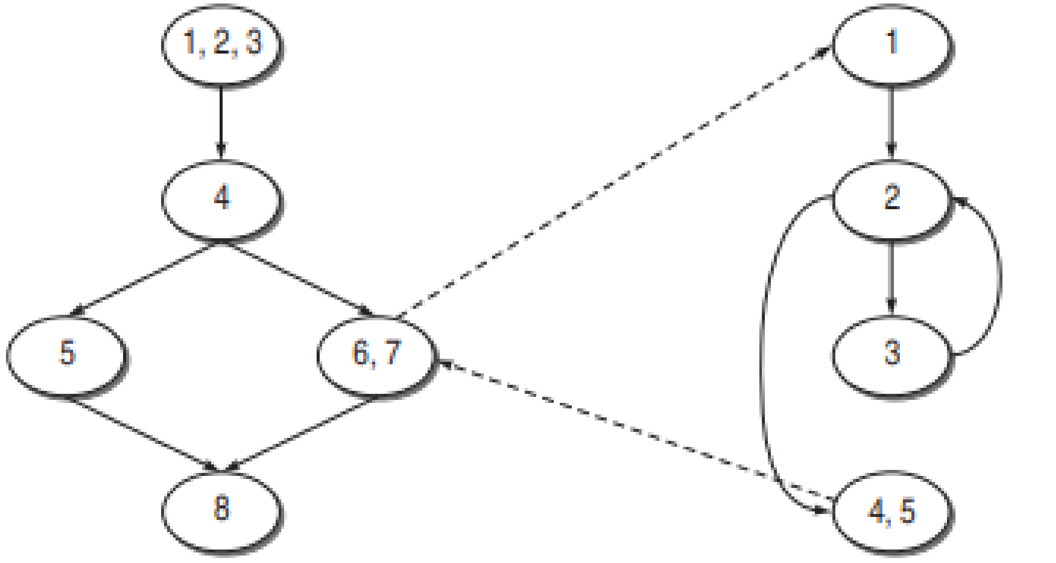
V (M) = e − n + 2p
= 5 – 5 + 2
= 2

Cyclomatic complexity of fact()

V (R) = e – n + 2p
= 4 − 4 + 2
= 2

Cyclomatic complexity of the whole graph considering the full program

(a) V (G) = e – n + 2p
= 9 − 9 + 2*2
= 4
= V (M) + V (R)



(a) Flow graph for main () (b) Flow graph for fact ()

Halstead’s Complexity

- Halstead’s Software metrics are a set of measures proposed by Maurice Halstead to evaluate the complexity of a software program.
- These metrics are based on the number of distinct operators and operands in the program
- This is used to estimate the effort required to develop and maintain the program.

Notations of Halstead Metrics

- n1 = Number of distinct operators.
- n2 = Number of distinct operands.
- N1 = Total number of occurrences of operators.
- N2 = Total number of occurrences of operands.

Notations of Halstead Metrics

- Program length (N): This is the total number of operator and operand occurrences in the program.
- Vocabulary size (n): This is the total number of distinct operators and operands in the program.
- Program volume (V): This is the product of program length (N) and the logarithm of vocabulary size (n), i.e., $V = N*\log_2(n)$.
- Program level (L): This is the ratio of the number of operator occurrences to the number of operand occurrences in the program, i.e., $L = n_1/n_2$, where n_1 is the number of operator occurrences and n_2 is the number of operand occurrences.

Notations of Halstead Metrics

- Program difficulty (D): This is the ratio of the number of unique operators to the total number of operators in the program, i.e., $D = (n_1 /2) * (N_2 / n_2)$.
- Program effort (E): This is the product of program volume (V) and program difficulty (D), i.e., $E = V*D$.
- Time to implement (T): This is the estimated time required to implement the program, based on the program effort (E) and a constant value that depends on the programming language and development environment.

Calculate the Metrics of the Code Snippet

```
if (x[i] < x[j])
        {
            Save = x[i];
            x[i] = x[j];
            x[j] = save;
        }
```

Calculation of N

List of total operator and operands of the code snippet
$N_1 = 24$
$N_2 = 14$
So, Program length, $N = 38$

Operator	Operator
If	=
(	[
[	]
]	;
<	[
[	]
]	=
)	;
{	}
=	
[	
]	
;	
[	
]	

Operand
x
i
x
j
save
x
i
x
i
x
i
x
j
save

Calculation of n

List of distinct operator and operands of the code snippet

$n_1 = 10$

$n_2 = 4$

So, program vocabulary, $n = 14$

However, estimated program length is, $\widehat{N} = n_1 \log_2(n_1) + n_2 \log_2(n_2)$

$= 10 \log_2(10) + 4\log_2(4) = 41.21$

Operator
If
(
[
]
<
)
{
=
;
}

Operand
x
i
j
save

Calculation

- Program volume (V) = $N*\log_2(n)$

  $= 38*\log_2(14) = 144.68.$

- Potential Minimum Volume: The potential minimum volume V* is defined as the volume of the most succinct program in which a problem can be coded, $V* = (2 + n_2*) * \log_2(2 + n_2*)$, n_2* is the count of unique input and output parameters.

  Here, $n_2* = 6$

  $V* = (2 + 6) * \log_2(2 + 6) = 24$

- Program level (L) = n_1 / n_2

  $= 10/4 = 2.5$

Calculation

- Program difficulty (D) = $(n_1/2) * (N_2 / n_2)$
  = (10/2)*(14/4) = 17.5
- Program effort (E) = V*D
  = 144.68*17.5 = 2531.9
- Program time (T) = E/β
  = 2531.3/18 = 140.63

  β is a constant whose value is often considered as 18.