Software Testing

Data Flow Graph Testing

By

Abhinandan Jain (MT2021004)

Sagar Shirke (MT2021112)

Naidu Siripurapu (MT2021137)

Contents

Overview

The aim of the project is to perform Data Flow Graph Testing on a scientific calculator code by finding DU pairs, DU paths, All DU path coverage and designing the test cases according to Data Flow Graph.

Source Code for Testing

- We have developed the console-based java application called Scientific Calculator.
- Scientific Calculator provide following features

  Basic Arithmetic Operations
    - Addition
    - Subtraction
    - Multiplication
    - Division
    - Modulo
    - Square
    - Square root
    - Multiplicative Inverse
    - Addition of array elements

  Advanced Arithmetic Operations
    - Log (Natural)
    - Log (base b)
    - Factorial
    - Permutations (nPr)
    - Combinations (nCr)
    - Calculate y th Power of x (x ^ y)

  Conversion between Number Systems
    - Convert a Roman Number to an Integer
    - Convert an Integer to a Roman Number
    - Decimal to Binary
    - Binary to Decimal
    - Binary to Hexadecimal

- Decimal to Hexadecimal
- Hexadecimal to Decimal
- Octal to Binary
- Binary to Octal
- Decimal to Octal
- Octal to Hexadecimal
- Hexadecimal to Octal

Calculate Area / Volume

- Area of Square.
- Area of Rectangle.
- Area of Circle.
- Area of Triangle.
- Total Surface Area of Cube.
- Total Surface Area of Cylinder.
- Total Surface Area of Cone.
- Total Surface Area of Sphere.
- Volume of Cube.
- Volume of Cylinder.
- Volume of Cone.
- Volume of Sphere.

Data Flow Graph

- We are creating Data Flow Graph (DFG) for each functionality.
- After creating Data Flow Graph, we found du pairs, du paths and all du path coverage.
- The tool used for finding du pairs, du paths and all du path coverage is https://cs.gmu.edu:8443/offutt/coverage/DFGraphCoverage
- Test cases generation and testing is done using JUnit.

Tools used:

- JUnit
- Data Flow Graph Coverage Web Application (https://cs.gmu.edu:8443/offutt/coverag/DFGraphCoverage)

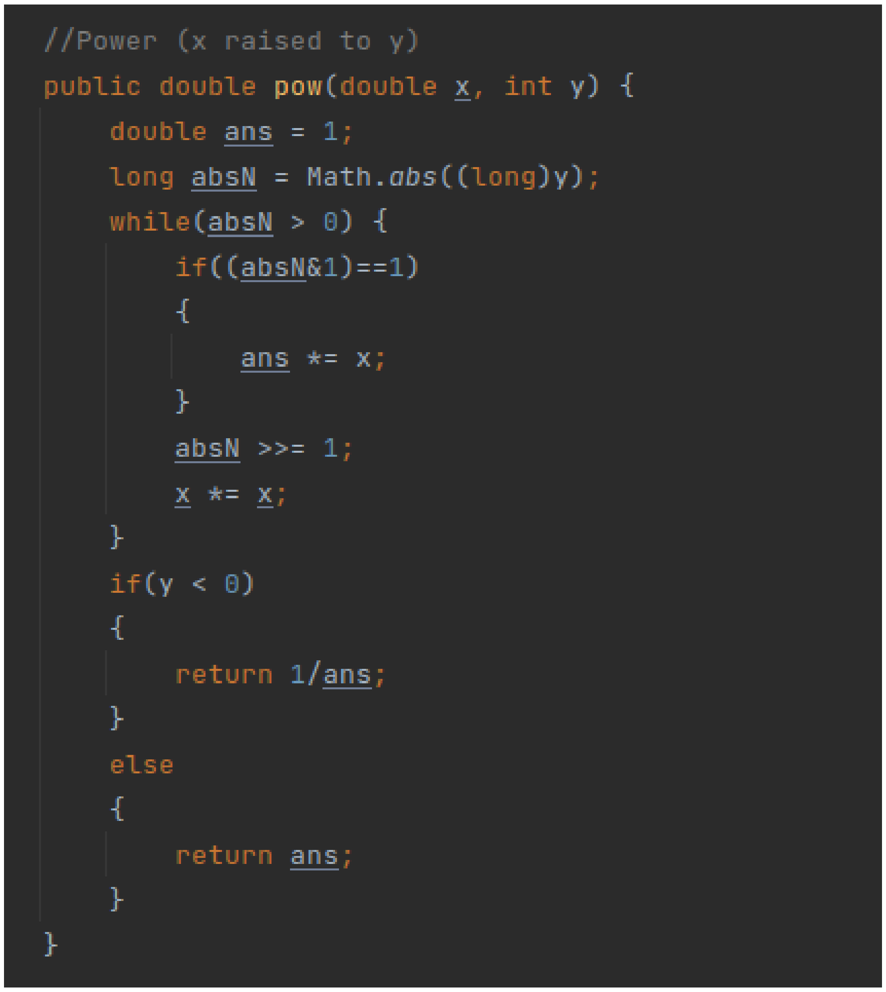
Functionalities along with Data Flow Graph Testing

Below We have mentioned some of the functions of Scientific Calculator along with the source code, Control Flow Graph, Data Flow Graph, du pairs, du paths and all du path coverage.

- ### pow(x,y)

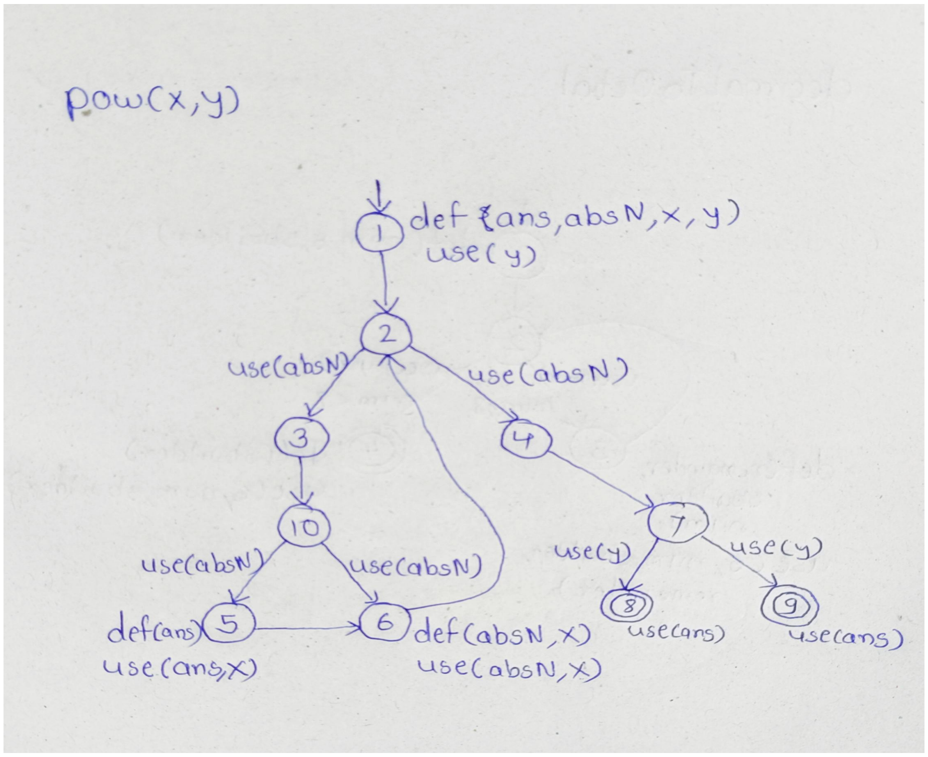
  - This function calculates yth power of x. (i.e. x^y)

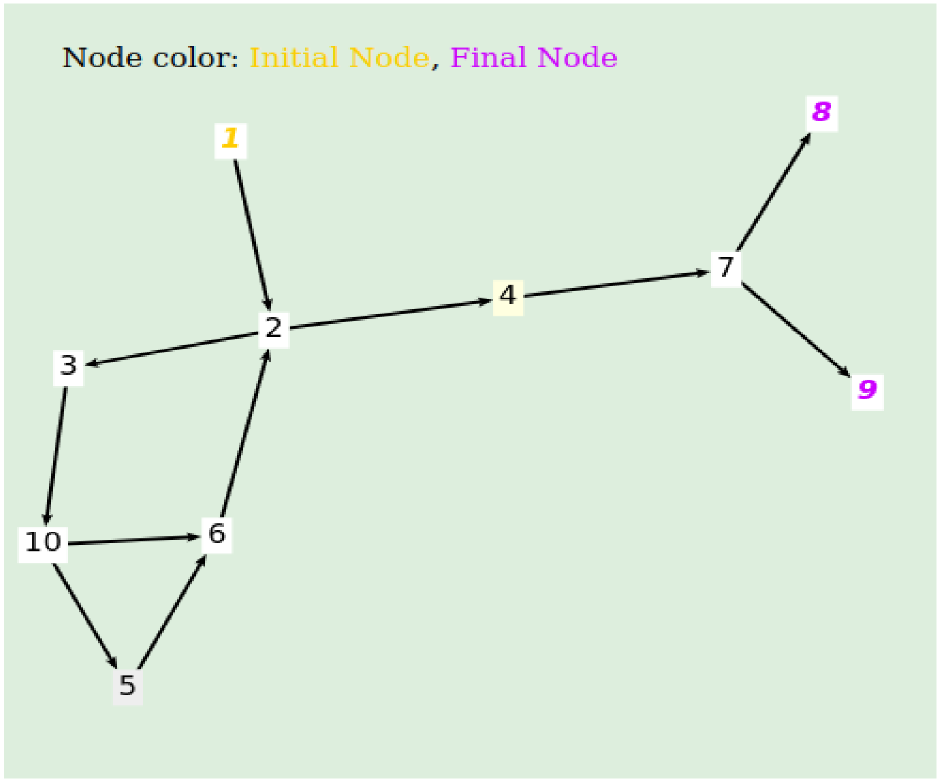
  - **Code:**



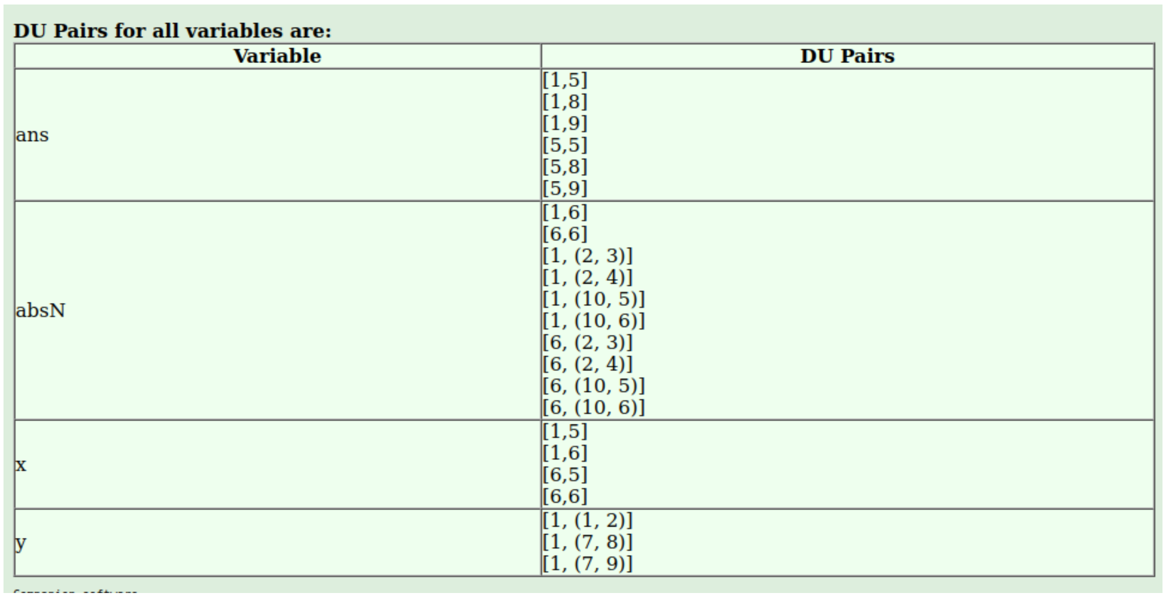
```java
//Power (x raised to y)
public double pow(double x, int y) {
    double ans = 1;
    long absN = Math.abs((long)y);
    while(absN > 0) {
        if((absN&1)==1)
        {
            ans *= x;
        }
        absN >>= 1;
        x *= x;
    }
    if(y < 0)
    {
        return 1/ans;
    }
    else
    {
        return ans;
    }
}
```

❖ Data Flow Graph:





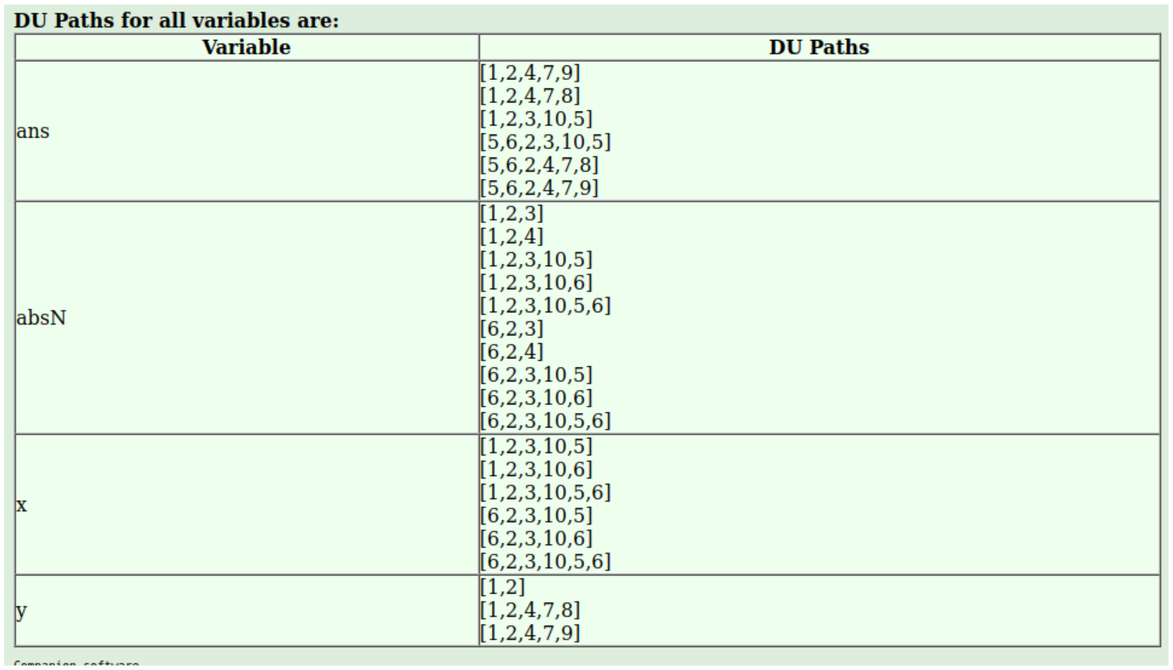
❖ DU pairs



DU Pairs for all variables are:

Variable	DU Pairs
ans	[1,5] [1,8] [1,9] [5,5] [5,8] [5,9]
absN	[1,6] [6,6] [1, (2, 3)] [1, (2, 4)] [1, (10, 5)] [1, (10, 6)] [6, (2, 3)] [6, (2, 4)] [6, (10, 5)] [6, (10, 6)]
x	[1,5] [1,6] [6,5] [6,6]
y	[1, (1, 2)] [1, (7, 8)] [1, (7, 9)]

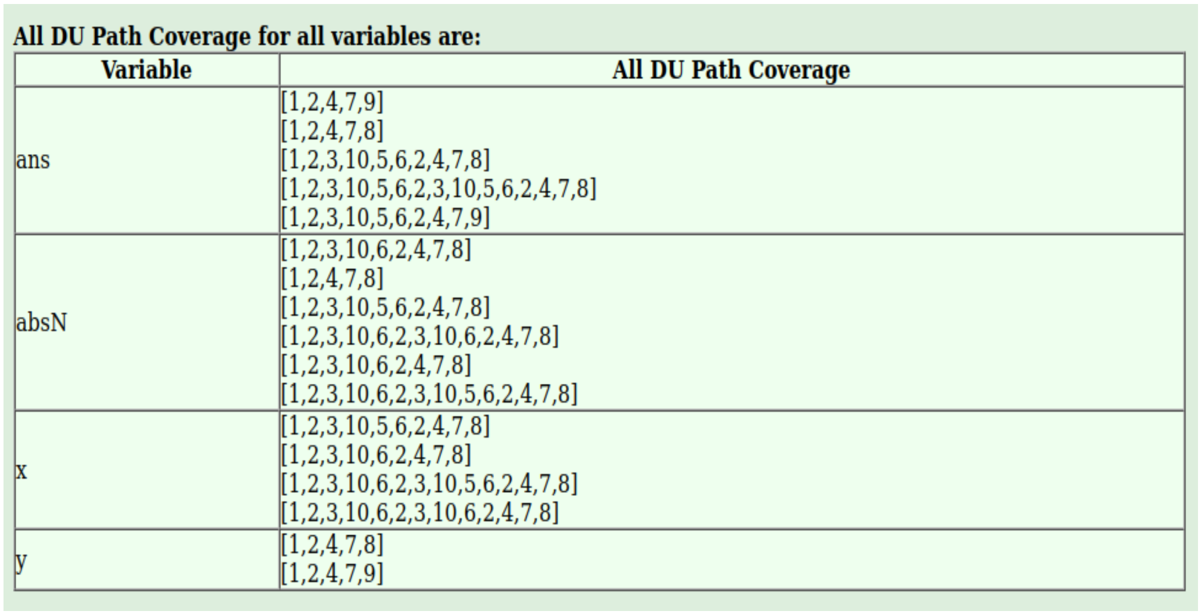
❖ DU paths



DU Paths for all variables are:

Variable	DU Paths
ans	[1,2,4,7,9] [1,2,4,7,8] [1,2,3,10,5] [5,6,2,3,10,5] [5,6,2,4,7,8] [5,6,2,4,7,9]
absN	[1,2,3] [1,2,4] [1,2,3,10,5] [1,2,3,10,6] [1,2,3,10,5,6] [6,2,3] [6,2,4] [6,2,3,10,5] [6,2,3,10,6] [6,2,3,10,5,6]
x	[1,2,3,10,5] [1,2,3,10,6] [1,2,3,10,5,6] [6,2,3,10,5] [6,2,3,10,6] [6,2,3,10,5,6]
y	[1,2] [1,2,4,7,8] [1,2,4,7,9]

❖ All DU Path Coverage



All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
ans	[1,2,4,7,9]
	[1,2,4,7,8]
	[1,2,3,10,5,6,2,4,7,8]
	[1,2,3,10,5,6,2,3,10,5,6,2,4,7,8]
	[1,2,3,10,5,6,2,4,7,9]
absN	[1,2,3,10,6,2,4,7,8]
	[1,2,4,7,8]
	[1,2,3,10,5,6,2,4,7,8]
	[1,2,3,10,6,2,3,10,6,2,4,7,8]
	[1,2,3,10,6,2,4,7,8]
	[1,2,3,10,6,2,3,10,5,6,2,4,7,8]
x	[1,2,3,10,5,6,2,4,7,8]
	[1,2,3,10,6,2,4,7,8]
	[1,2,3,10,6,2,3,10,5,6,2,4,7,8]
	[1,2,3,10,6,2,3,10,6,2,4,7,8]
y	[1,2,4,7,8]
	[1,2,4,7,9]

There are total 7 unique paths:

1. That skip the loop:

- [1,2,4,7,9], [1,2,4,7,8]
  - o Test case for [1,2,4,7,9]:
    - pow(x,y)⇒ pow(2,0)
    - Expected Output = 1
  - o Test case for [1,2,4,7,8]:
    - Infeasible to cover this case because if y is -Ve then it will go inside the loop, only one way to execute this code without loop is y==0 but then condition y<0 will never true. Hence, we cannot reach node 8 without execution loop.

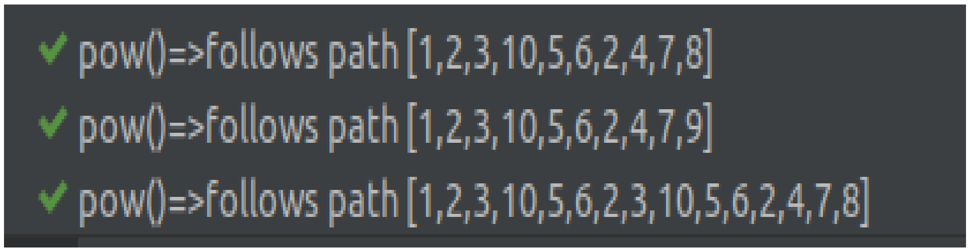
2. That requires at least one iteration of loop:

  - ▪ [1,2,3,10,5,6,2,4,7,8], [1,2,3,10,5,6,2,4,7,9], [1,2,3,10,6,2,4,7,8]
    - o Test case for [1,2,3,10,5,6,2,4,7,8]:
      - pow(x,y)⇒ pow(2,-1)
      - Expected output = 0.5
    - o Test case for [1,2,3,10,5,6,2,4,7,9]:
      - pow(x,y)⇒ pow(2,1)
      - Expected output = 2

- Test case for [1,2,3,10,6,2,4,7,8]:
  - Again, this case is infeasible to cover because if (absN&1)≠1 then it requires one more iteration.

3. That requires at least two iterations of loop:
   - [1,2,3,10,5,6,2,3,10,5,6,2,4,7,8], [1,2,3,10,5,6,2,3,10,6,2,4,7,8]
     - Test case for [1,2,3,10,5,6,2,3,10,5,6,2,4,7,8]:
       - Pow(x,y)⇒ pow(2,-3)
       - Expected output = 0.125
     - Test case for [1,2,3,10,5,6,2,3,10,5,6,2,4,7,8]:
       - once again, this case is infeasible to cover because of the same reason above if (absN&1)≠1 then it requires one more iteration.
       -

❖ <u>Result:</u>

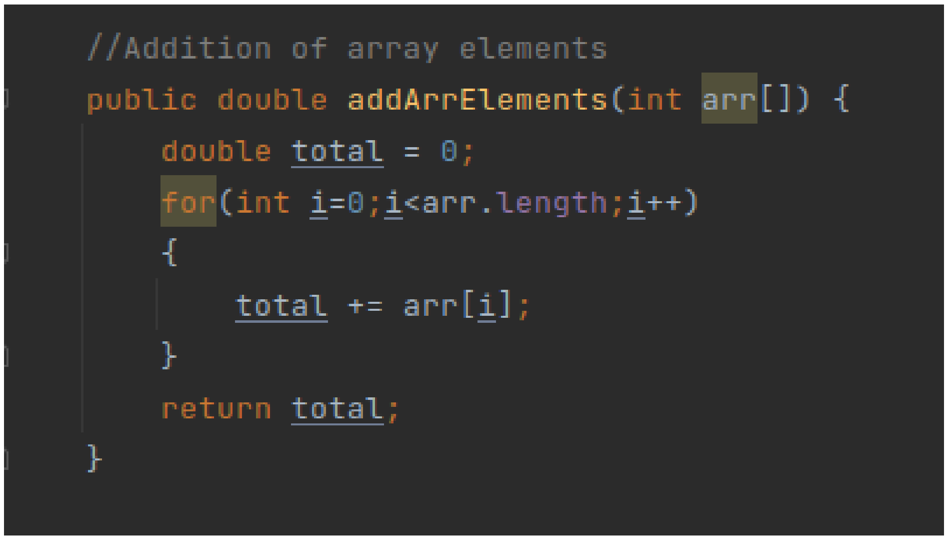


```
✓ pow()=>follows path [1,2,3,10,5,6,2,4,7,8]
✓ pow()=>follows path [1,2,3,10,5,6,2,4,7,9]
✓ pow()=>follows path [1,2,3,10,5,6,2,3,10,5,6,2,4,7,8]
```

♦ addArrElements()

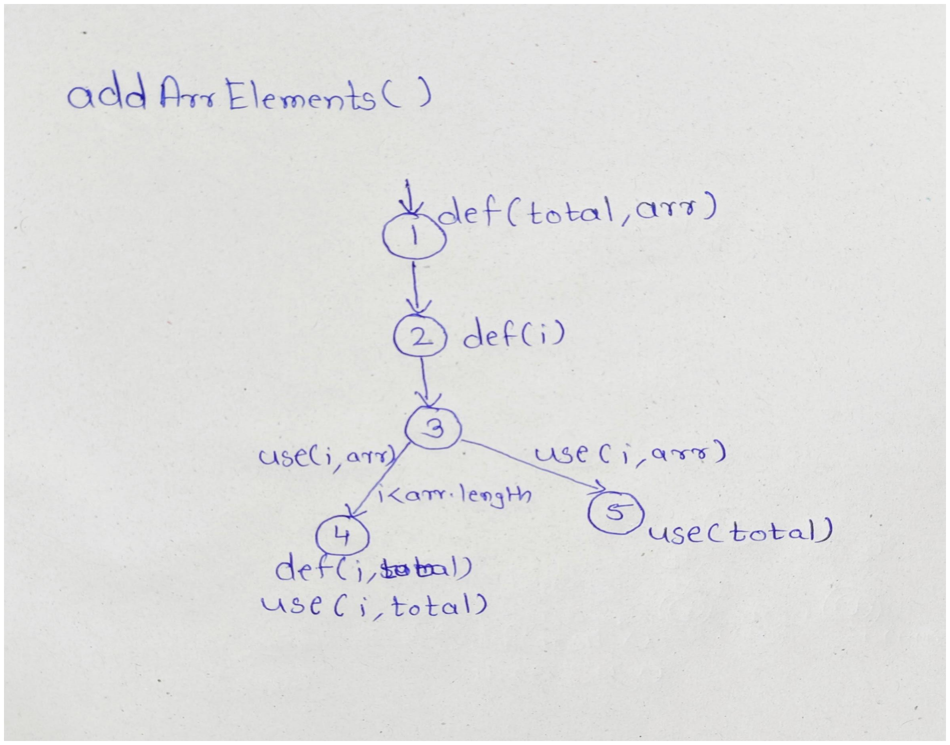
- ❖ This function is used to find the sum of array elements

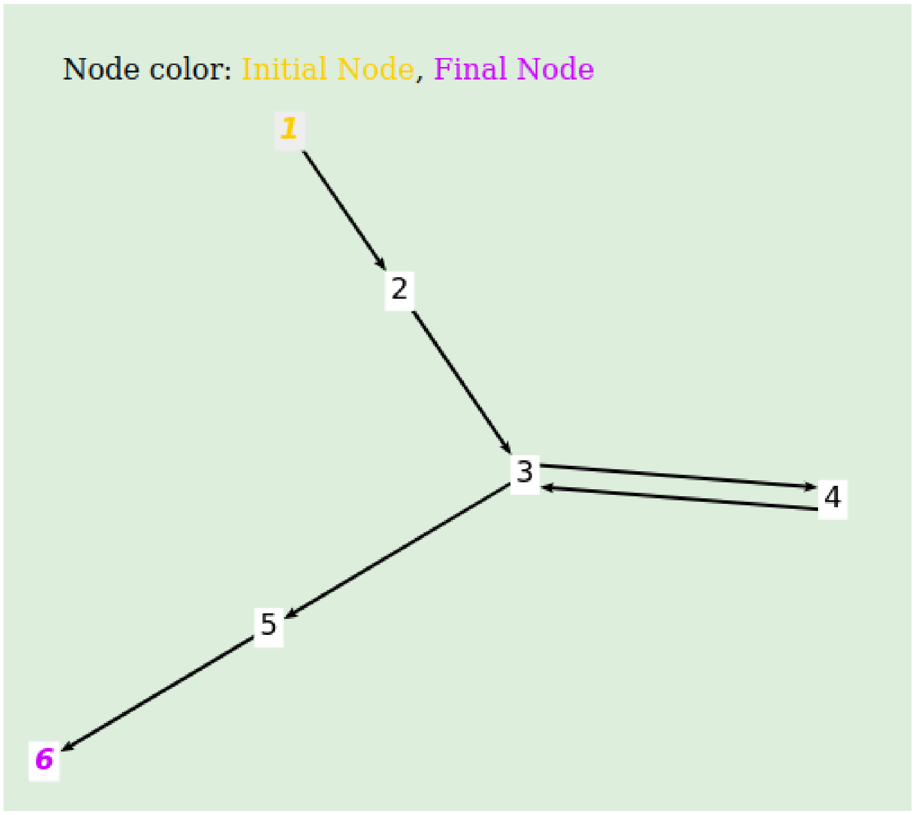
❖ Code:



```
//Addition of array elements
public double addArrElements(int arr[]) {
    double total = 0;
    for(int i=0;i<arr.length;i++)
    {
        total += arr[i];
    }
    return total;
}
```

❖ Data Flow Graph



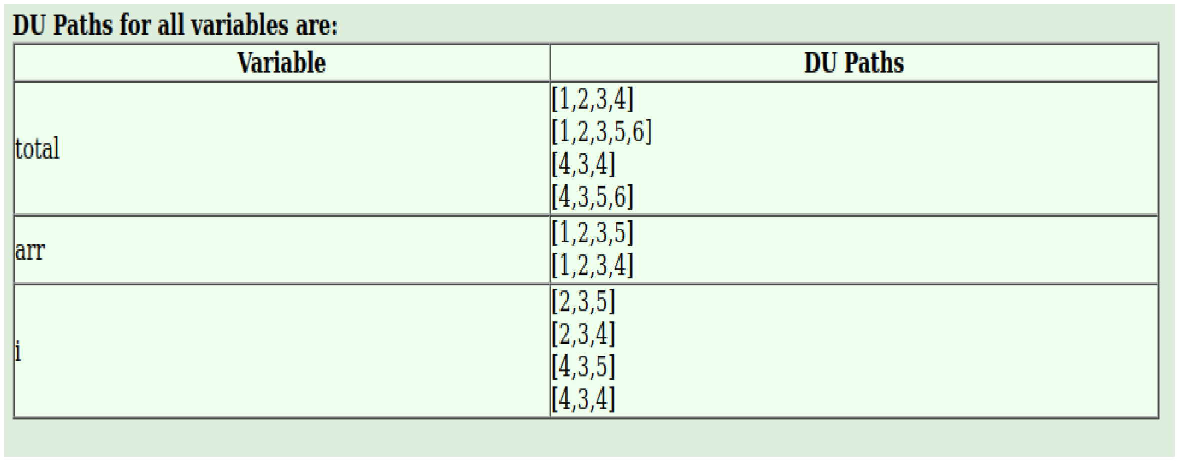


❖ DU pairs:

DU Pairs for all variables are:

Variable	DU Pairs
total	[1,6] [1,4] [4,6] [4,4]
arr	[1, (3, 4)] [1, (3, 5)]
i	[2,4] [4,4] [2, (3, 4)] [2, (3, 5)] [4, (3, 4)] [4, (3, 5)]

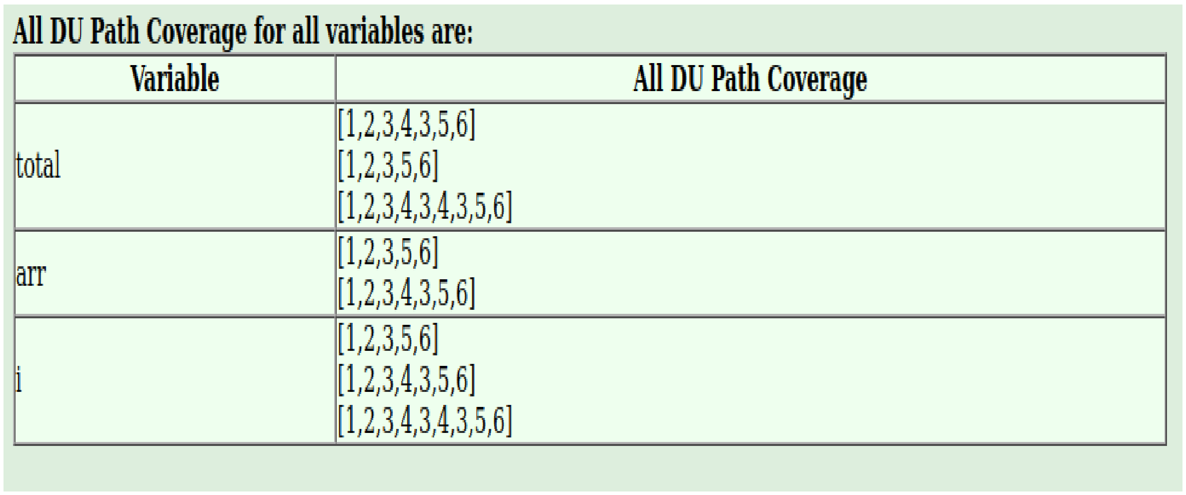
❖ DU paths:



DU Paths for all variables are:

Variable	DU Paths
total	[1,2,3,4] [1,2,3,5,6] [4,3,4] [4,3,5,6]
arr	[1,2,3,5] [1,2,3,4]
i	[2,3,5] [2,3,4] [4,3,5] [4,3,4]

❖ All DU path coverage:



All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
total	[1,2,3,4,3,5,6] [1,2,3,5,6] [1,2,3,4,3,4,3,5,6]
arr	[1,2,3,5,6] [1,2,3,4,3,5,6]
i	[1,2,3,5,6] [1,2,3,4,3,5,6] [1,2,3,4,3,4,3,5,6]

There are 3 unique paths

1. One that skips the loop
    a. [1,2,3,5,6]
        i. Test case for this is ⇒ fact(arr1) //where arr1 is array with 0 element
        Expected output = 0
2. One that iterate loop 1 time
    a. [1,2,3,4,3,5,6]
        i. Test case for this is ⇒ fact(arr2) //where arr2 is array with 1 element
        Expected output = 1

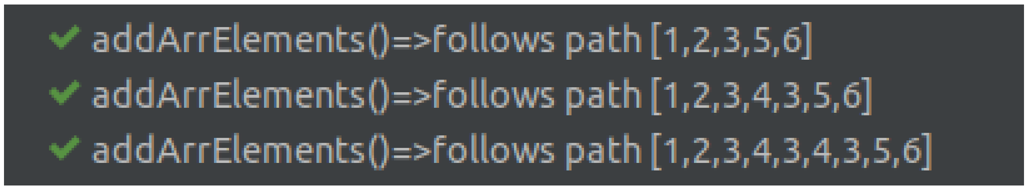
3. That iterate loop 2 or more times

a. [1,2,3,4,3,4,3,5,6]

i. Test case for this is ⇒ fact(arr3) //where arr3 is array with 3 elements

Expected output = 10

❖ <u>Result:</u>

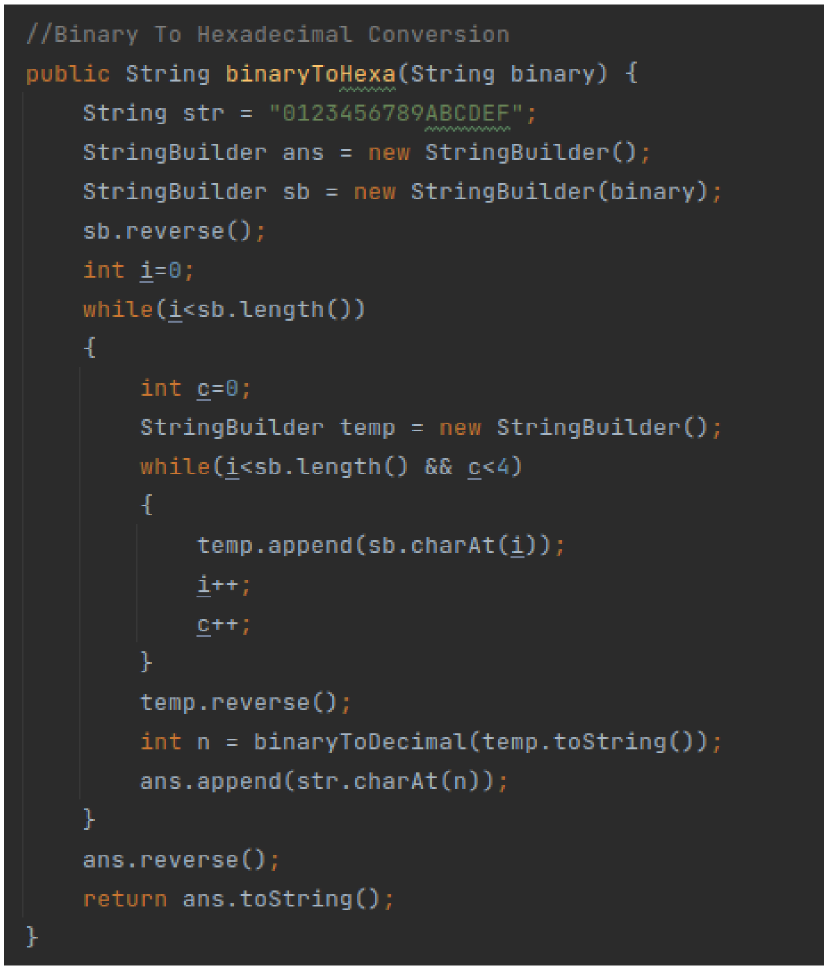


```
✓ addArrElements()=>follows path [1,2,3,5,6]
✓ addArrElements()=>follows path [1,2,3,4,3,5,6]
✓ addArrElements()=>follows path [1,2,3,4,3,4,3,5,6]
```

♦ binaryToHexa()

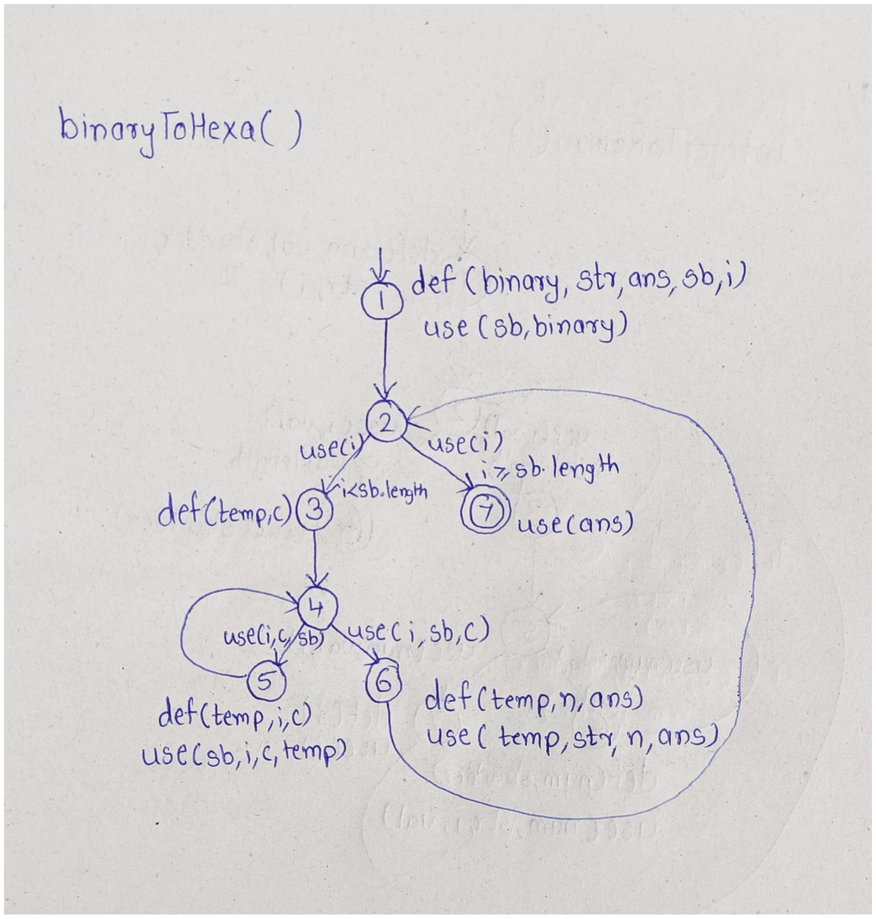
❖ This function converts binary number to its hexadecimal equivalent format.

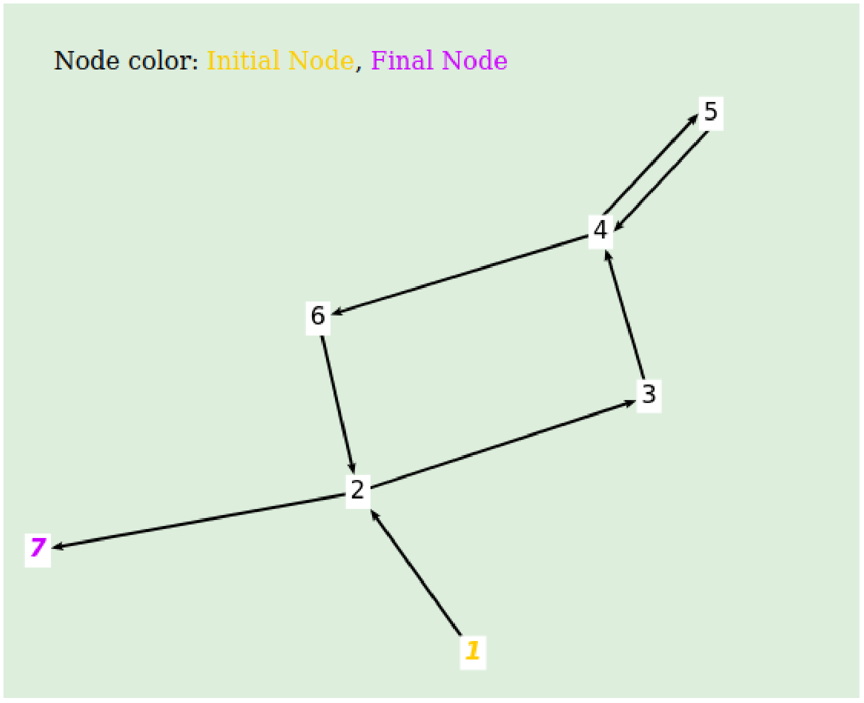
❖ <u>Code:</u>



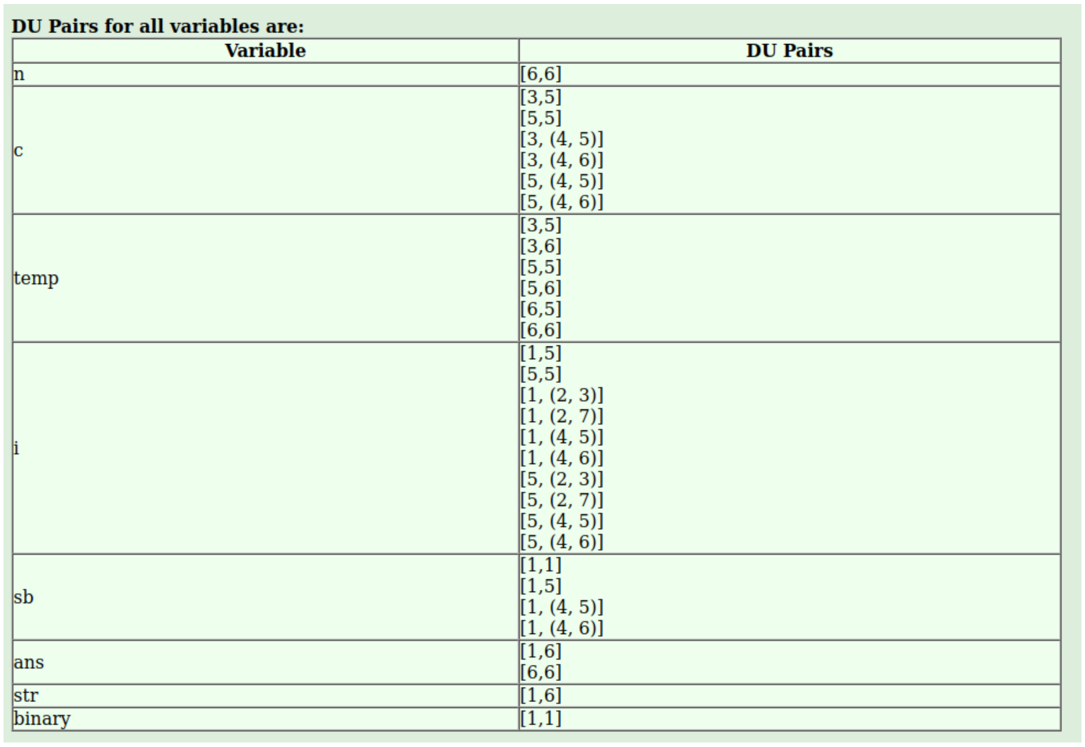
```java
//Binary To Hexadecimal Conversion
public String binaryToHexa(String binary) {
    String str = "0123456789ABCDEF";
    StringBuilder ans = new StringBuilder();
    StringBuilder sb = new StringBuilder(binary);
    sb.reverse();
    int i=0;
    while(i<sb.length())
    {
        int c=0;
        StringBuilder temp = new StringBuilder();
        while(i<sb.length() && c<4)
        {
            temp.append(sb.charAt(i));
            i++;
            c++;
        }
        temp.reverse();
        int n = binaryToDecimal(temp.toString());
        ans.append(str.charAt(n));
    }
    ans.reverse();
    return ans.toString();
}
```

❖ Data Flow Graph:





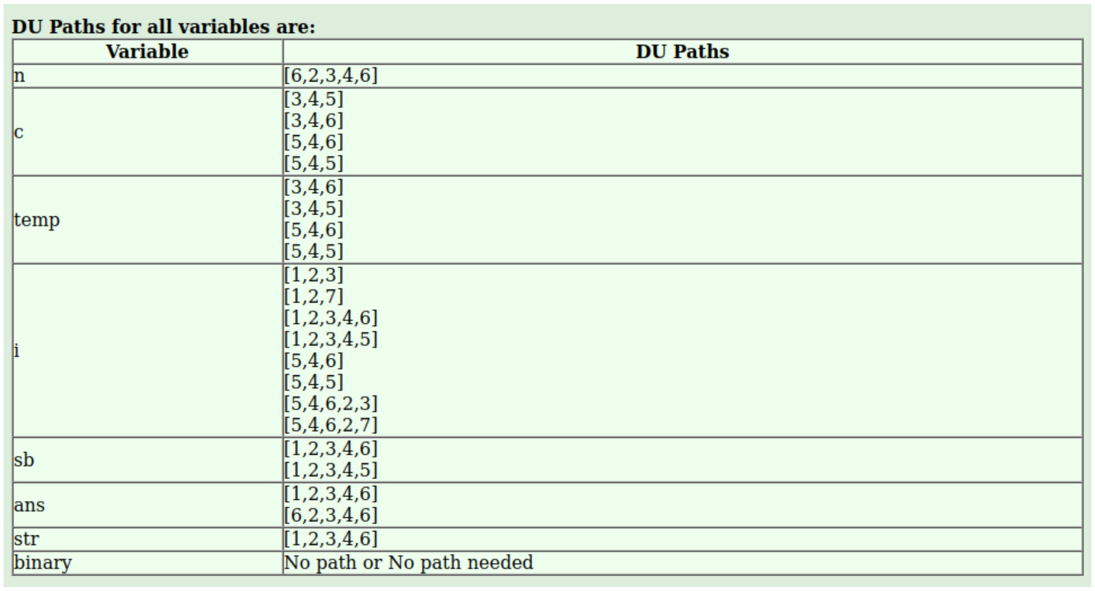
❖ Du pairs:



DU Pairs for all variables are:

Variable	DU Pairs
n	[6,6]
c	[3,5] [5,5] [3, (4, 5)] [3, (4, 6)] [5, (4, 5)] [5, (4, 6)]
temp	[3,5] [3,6] [5,5] [5,6] [6,5] [6,6]
i	[1,5] [5,5] [1, (2, 3)] [1, (2, 7)] [1, (4, 5)] [1, (4, 6)] [5, (2, 3)] [5, (2, 7)] [5, (4, 5)] [5, (4, 6)]
sb	[1,1] [1,5] [1, (4, 5)] [1, (4, 6)]
ans	[1,6] [6,6]
str	[1,6]
binary	[1,1]

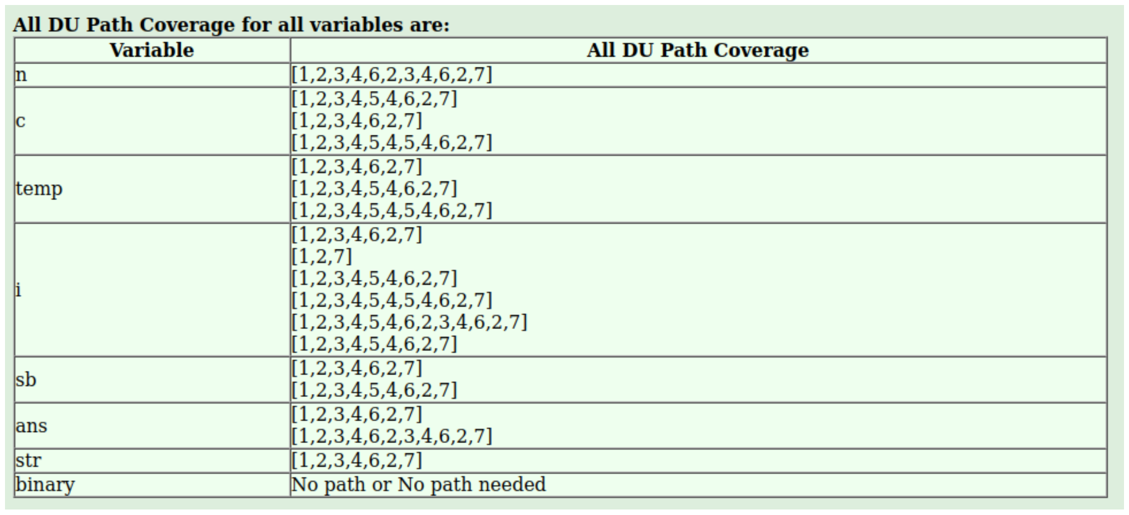
❖ Du Paths:



DU Paths for all variables are:

Variable	DU Paths
n	[6,2,3,4,6]
c	[3,4,5] [3,4,6] [5,4,6] [5,4,5]
temp	[3,4,6] [3,4,5] [5,4,6] [5,4,5]
i	[1,2,3] [1,2,7] [1,2,3,4,6] [1,2,3,4,5] [5,4,6] [5,4,5] [5,4,6,2,3] [5,4,6,2,7]
sb	[1,2,3,4,6] [1,2,3,4,5]
ans	[1,2,3,4,6] [6,2,3,4,6]
str	[1,2,3,4,6]
binary	No path or No path needed

❖ All Du Path Coverage



All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
n	[1,2,3,4,6,2,3,4,6,2,7]
c	[1,2,3,4,5,4,6,2,7] [1,2,3,4,6,2,7] [1,2,3,4,5,4,5,4,6,2,7]
temp	[1,2,3,4,6,2,7] [1,2,3,4,5,4,6,2,7] [1,2,3,4,5,4,5,4,6,2,7]
i	[1,2,3,4,6,2,7] [1,2,7] [1,2,3,4,5,4,6,2,7] [1,2,3,4,5,4,5,4,6,2,7] [1,2,3,4,5,4,6,2,3,4,6,2,7] [1,2,3,4,5,4,6,2,7]
sb	[1,2,3,4,6,2,7] [1,2,3,4,5,4,6,2,7]
ans	[1,2,3,4,6,2,7] [1,2,3,4,6,2,3,4,6,2,7]
str	[1,2,3,4,6,2,7]
binary	No path or No path needed

Here we have 6 unique paths:

1. That skips all the loop: [1,2,7]

- Test case for [1,2,7]:
  - binaryToHexa(String binary)⇒ binaryToHexa("") (An empty String)
  - Expected output = ""

2. That skip the second loop: [1,2,3,4,6,2,7], [1,2,3,4,6,2,3,4,6,2,7]

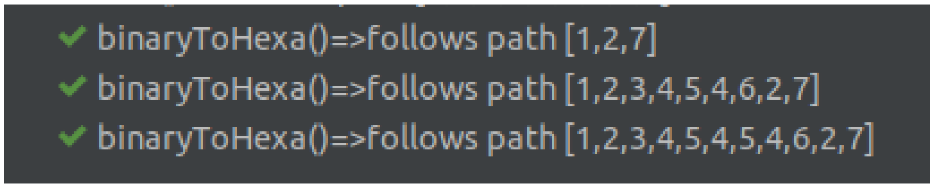
- Test case for [1,2,3,4,6,2,7]:
  - This case is infeasible to cover because if first while condition is true (i<sb.length()) then the second while condition (i<sb.length() & c<4) will always be true for first time.
- Test case for [1,2,3,4,6,2,3,4,6,2,7]:
  - Again, for the same reason this case is also infeasible.

3. That considers both the loop: [1,2,3,4,5,4,6,2,7], [1,2,3,4,5,4,5,4,6,2,7], [1,2,3,4,5,4,6,2,3,4,6,2,7]

- Test case for [1,2,3,4,5,4,6,2,7]:
  - binaryToHexa(String binary)⇒ binaryToHexa("1")
  - Expected output = "1"
- Test case for [1,2,3,4,5,4,5,4,6,2,7]:
  - binaryToHexa(String binary)⇒ binaryToHexa("10")
  - Expected output = "2"

- Test case for [1,2,3,4,5,4,6,2,3,4,6,2,7]:
    - This path is infeasible to cover because of the same reason again if the second loop condition becomes false after first iteration then first loop condition will always become false. so 4,5,5,6,2,3,2,4 this path will never be possible.

❖ Result

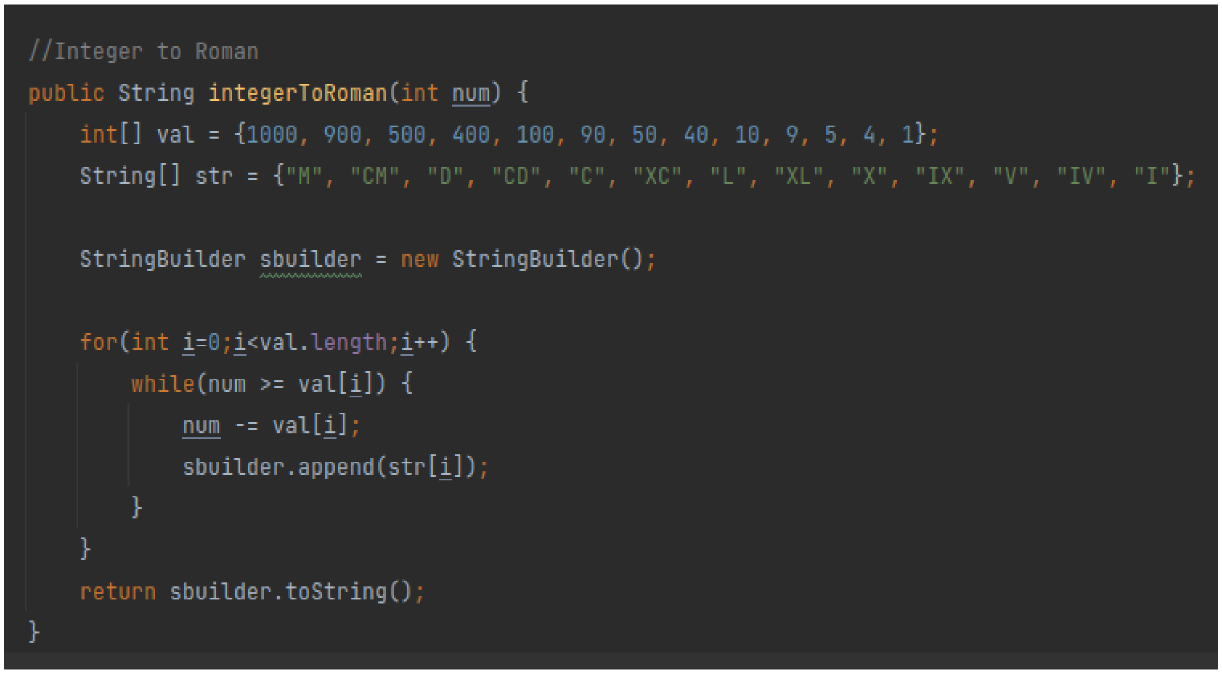


```
✓ binaryToHexa()=>follows path [1,2,7]
✓ binaryToHexa()=>follows path [1,2,3,4,5,4,6,2,7]
✓ binaryToHexa()=>follows path [1,2,3,4,5,4,5,4,6,2,7]
```

♦ integerToRoman()

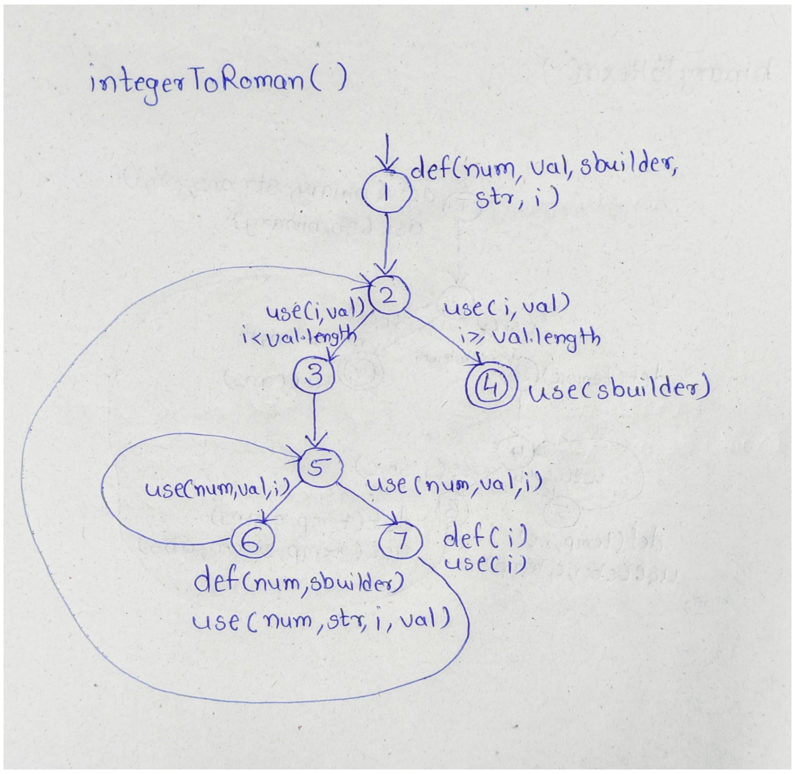
❖ This function gives roman equivalent of the integer which is given as a input to the program.

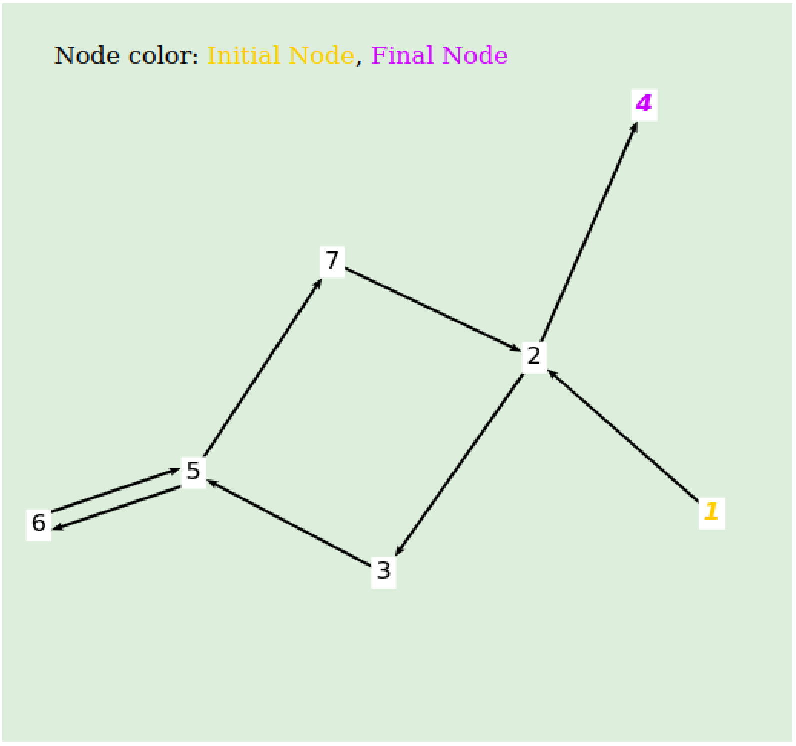
❖ Code:



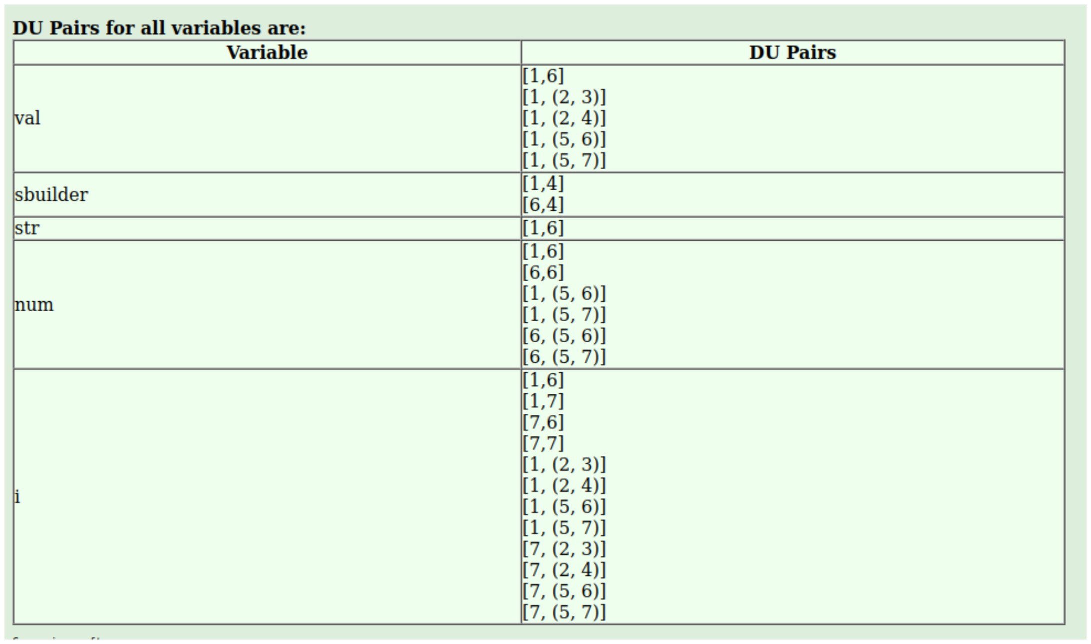
```java
//Integer to Roman
public String integerToRoman(int num) {
    int[] val = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
    String[] str = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};

    StringBuilder sbuilder = new StringBuilder();

    for(int i=0;i<val.length;i++) {
        while(num >= val[i]) {
            num -= val[i];
            sbuilder.append(str[i]);
        }
    }
    return sbuilder.toString();
}
```

❖ Data Flow Graph:





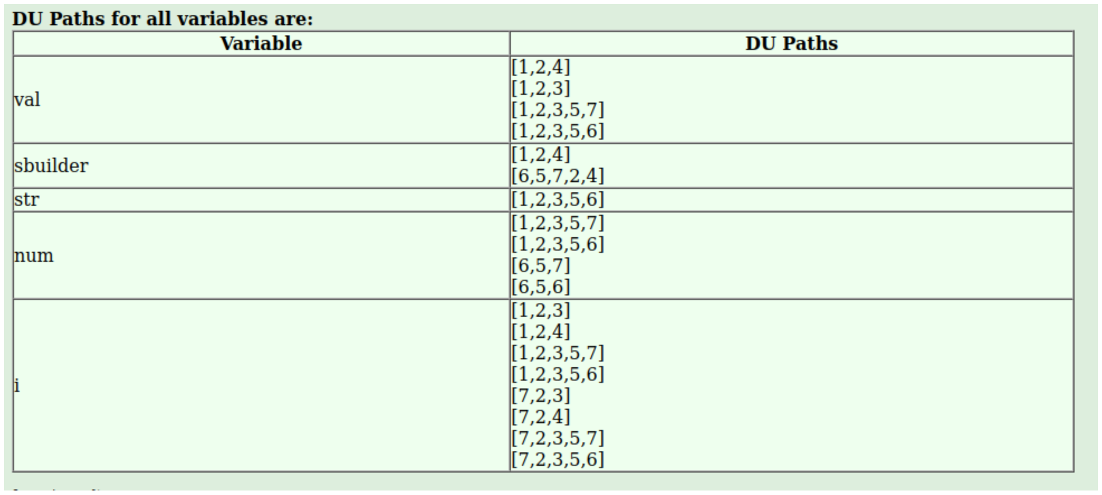
❖ DU pairs



DU Pairs for all variables are:

Variable	DU Pairs
val	[1,6] [1, (2, 3)] [1, (2, 4)] [1, (5, 6)] [1, (5, 7)]
sbuilder	[1,4] [6,4]
str	[1,6]
num	[1,6] [6,6] [1, (5, 6)] [1, (5, 7)] [6, (5, 6)] [6, (5, 7)]
i	[1,6] [1,7] [7,6] [7,7] [1, (2, 3)] [1, (2, 4)] [1, (5, 6)] [1, (5, 7)] [7, (2, 3)] [7, (2, 4)] [7, (5, 6)] [7, (5, 7)]

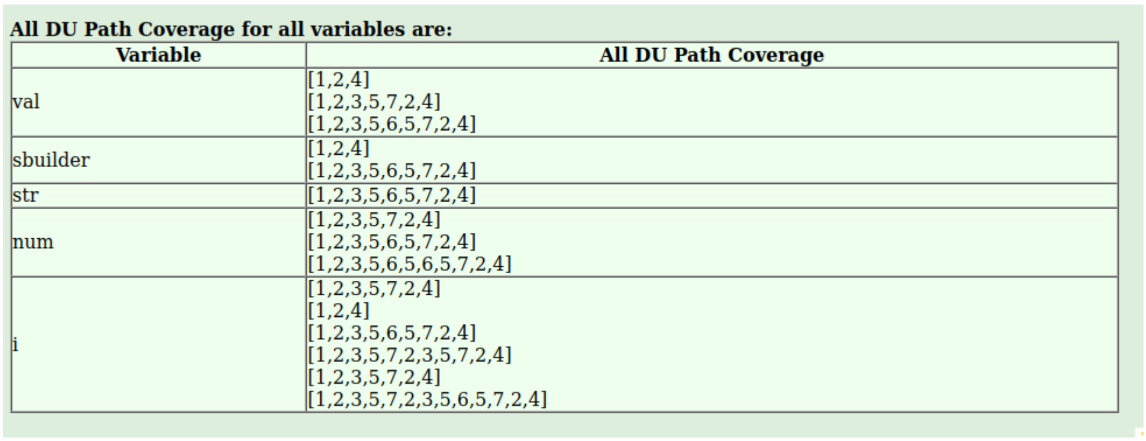
❖ DU paths



DU Paths for all variables are:

Variable	DU Paths
val	[1,2,4] [1,2,3] [1,2,3,5,7] [1,2,3,5,6]
sbuilder	[1,2,4] [6,5,7,2,4]
str	[1,2,3,5,6]
num	[1,2,3,5,7] [1,2,3,5,6] [6,5,7] [6,5,6]
i	[1,2,3] [1,2,4] [1,2,3,5,7] [1,2,3,5,6] [7,2,3] [7,2,4] [7,2,3,5,7] [7,2,3,5,6]

❖ All Du path coverage



All DU Path Coverage for all variables are:

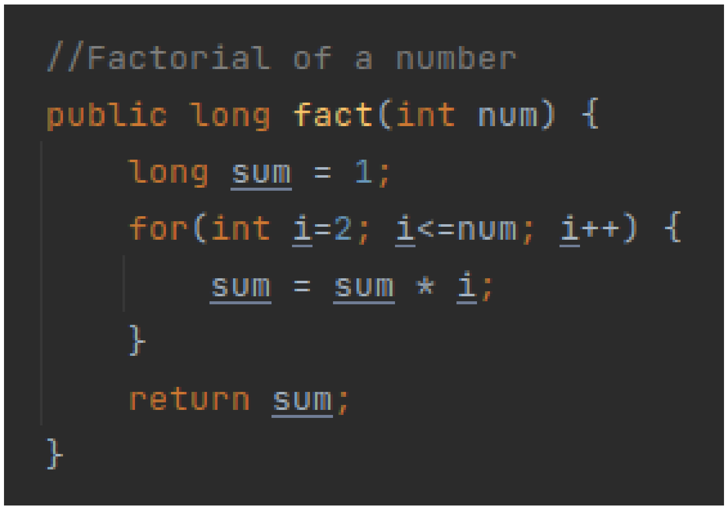
Variable	All DU Path Coverage
val	[1,2,4] [1,2,3,5,7,2,4] [1,2,3,5,6,5,7,2,4]
sbuilder	[1,2,4] [1,2,3,5,6,5,7,2,4]
str	[1,2,3,5,6,5,7,2,4]
num	[1,2,3,5,7,2,4] [1,2,3,5,6,5,7,2,4] [1,2,3,5,6,5,6,5,7,2,4]
i	[1,2,3,5,7,2,4] [1,2,4] [1,2,3,5,6,5,7,2,4] [1,2,3,5,7,2,3,5,7,2,4] [1,2,3,5,7,2,4] [1,2,3,5,7,2,3,5,6,5,7,2,4]

- There are total 5 unique paths: [1,2,4], [1,2,3,5,7,2,4], [1,2,3,5,6,5,7,2,4],[1,2,3,5,6,5,6,5,7,2,4], [1,2,3,5,7,2,3,5,6,5,7,2,4]
- These all paths are infeasible to cover because the outer for loop always requires 13 iterations because of length of the values array is 13.

♦ **fact()**

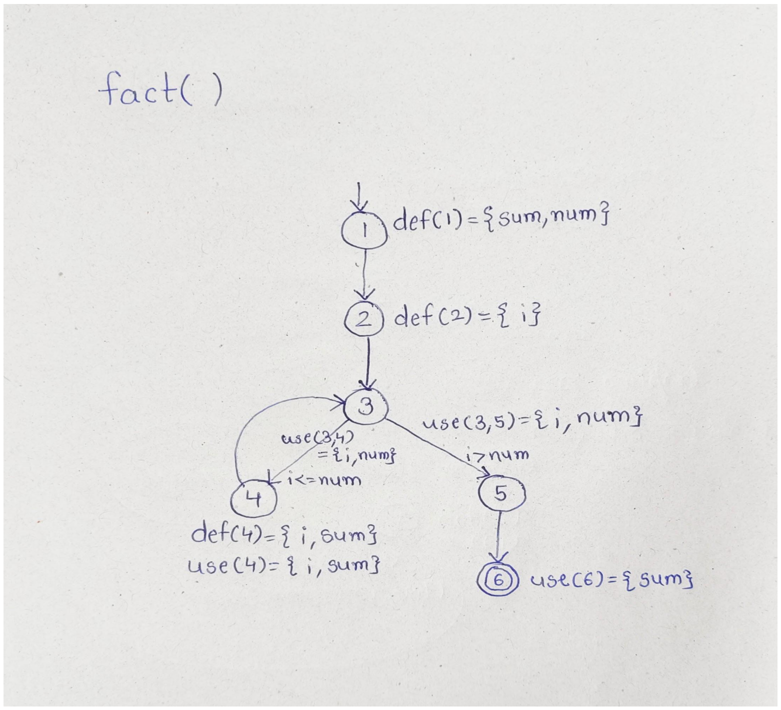
❖ This function returns the factorial of a given number.

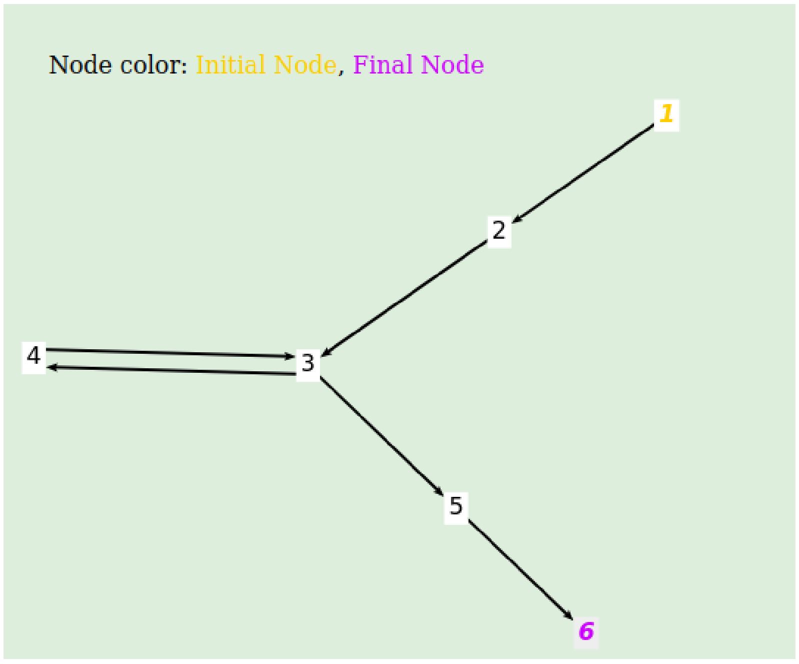
❖ Code:



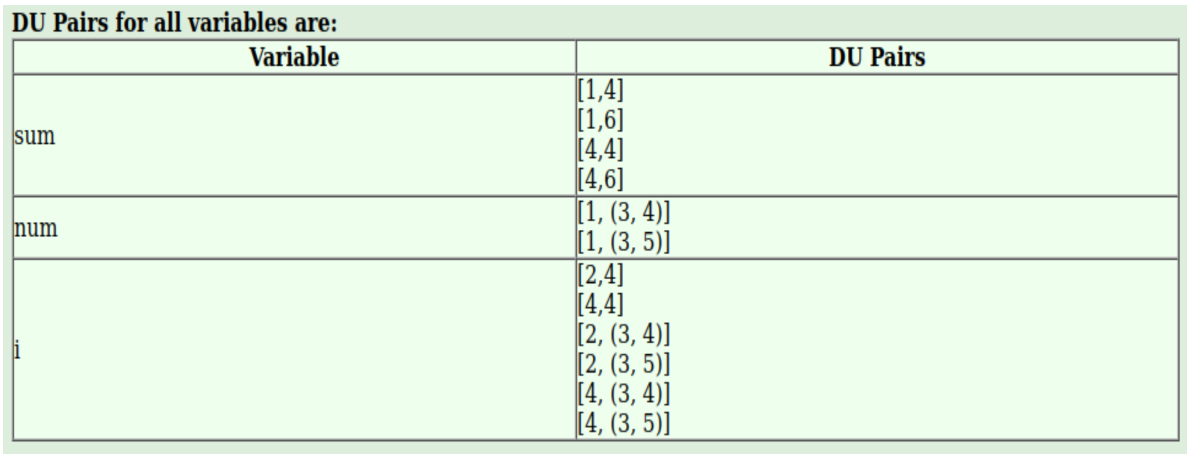
```java
//Factorial of a number
public long fact(int num) {
    long sum = 1;
    for(int i=2; i<=num; i++) {
        sum = sum * i;
    }
    return sum;
}
```

❖ Data Flow Graph:





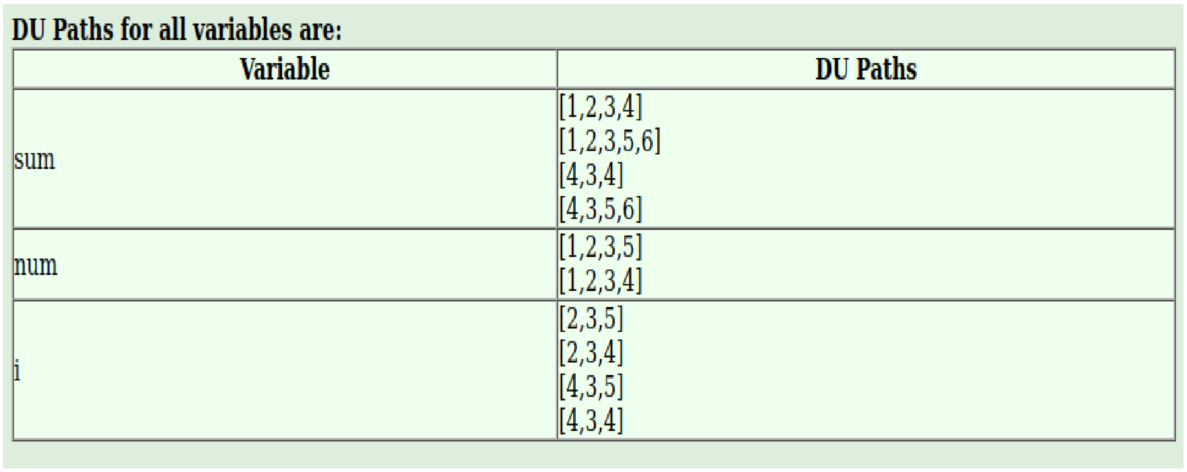
❖ Du pairs:



DU Pairs for all variables are:

Variable	DU Pairs
sum	[1,4] [1,6] [4,4] [4,6]
num	[1, (3, 4)] [1, (3, 5)]
i	[2,4] [4,4] [2, (3, 4)] [2, (3, 5)] [4, (3, 4)] [4, (3, 5)]

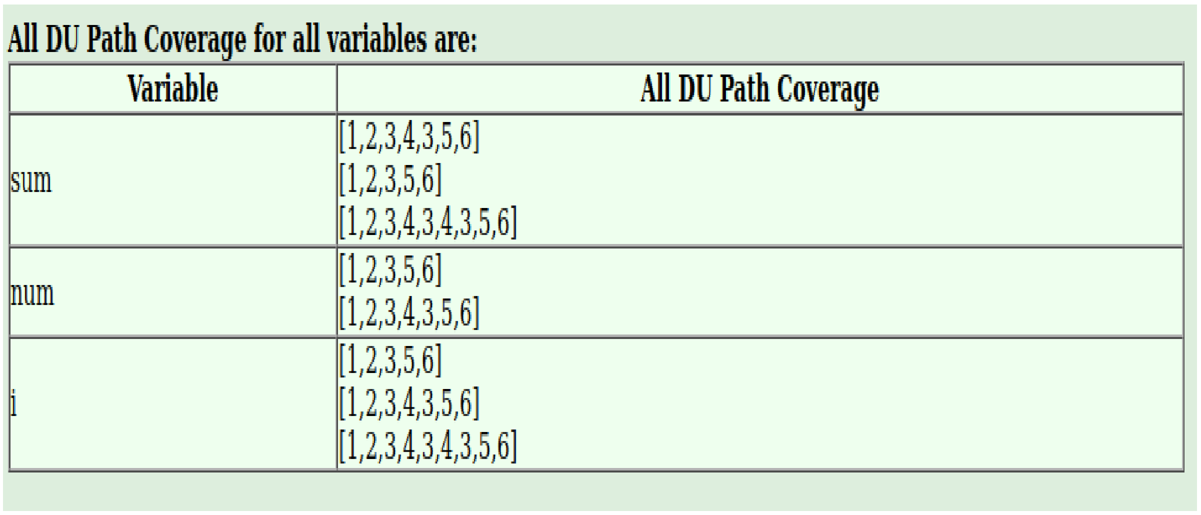
❖ Du Paths:



DU Paths for all variables are:

Variable	DU Paths
sum	[1,2,3,4] [1,2,3,5,6] [4,3,4] [4,3,5,6]
num	[1,2,3,5] [1,2,3,4]
i	[2,3,5] [2,3,4] [4,3,5] [4,3,4]

❖ All Du Path Coverage



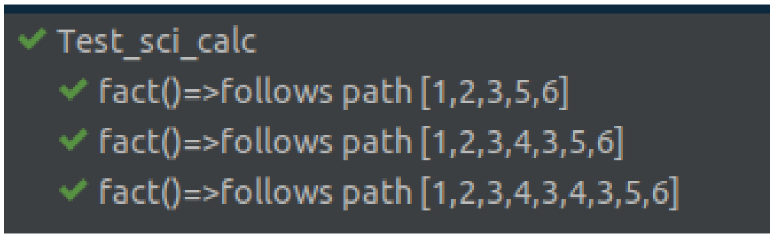
All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
sum	[1,2,3,4,3,5,6] [1,2,3,5,6] [1,2,3,4,3,4,3,5,6]
num	[1,2,3,5,6] [1,2,3,4,3,5,6]
i	[1,2,3,5,6] [1,2,3,4,3,5,6] [1,2,3,4,3,4,3,5,6]

There are 3 unique paths

1. One that skips the loop
    a. [1,2,3,5,6]
        i. Test case for this is ⇒ fact(1)
        Expected output = 1
2. One that iterate loop 1 time
    a. [1,2,3,4,3,5,6]
        i. Test case for this is ⇒ fact(2)
        Expected output = 2
3. That iterate loop 2 or more times
    a. [1,2,3,4,3,4,3,5,6]
        i. Test case for this is ⇒ fact(5)
        Expected output = 120

❖ Result:

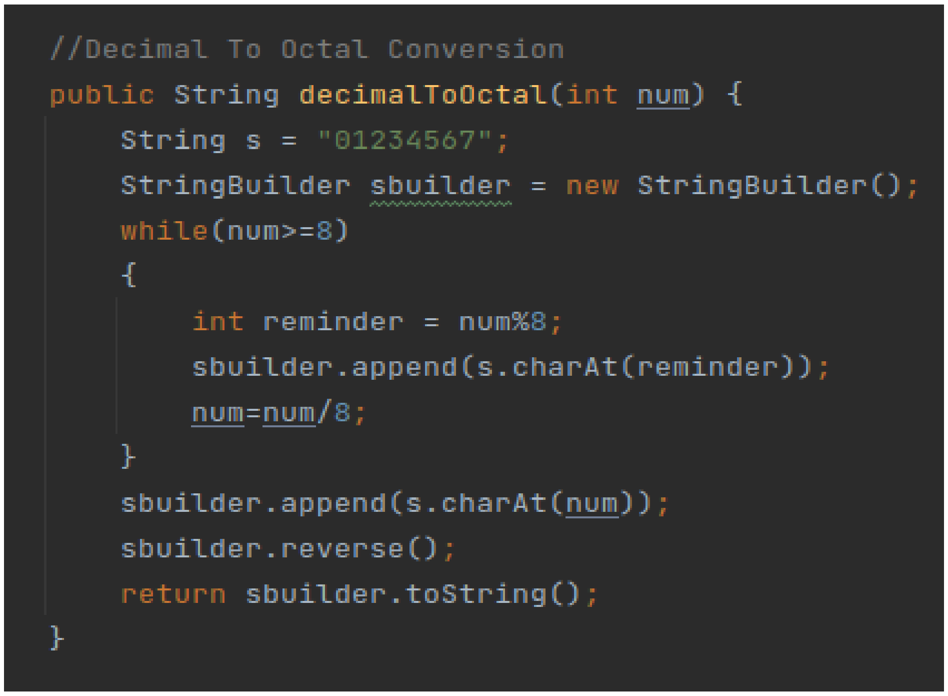


```
✓ Test_sci_calc
  ✓ fact()=>follows path [1,2,3,5,6]
  ✓ fact()=>follows path [1,2,3,4,3,5,6]
  ✓ fact()=>follows path [1,2,3,4,3,4,3,5,6]
```

♦ DecimalToOctal()

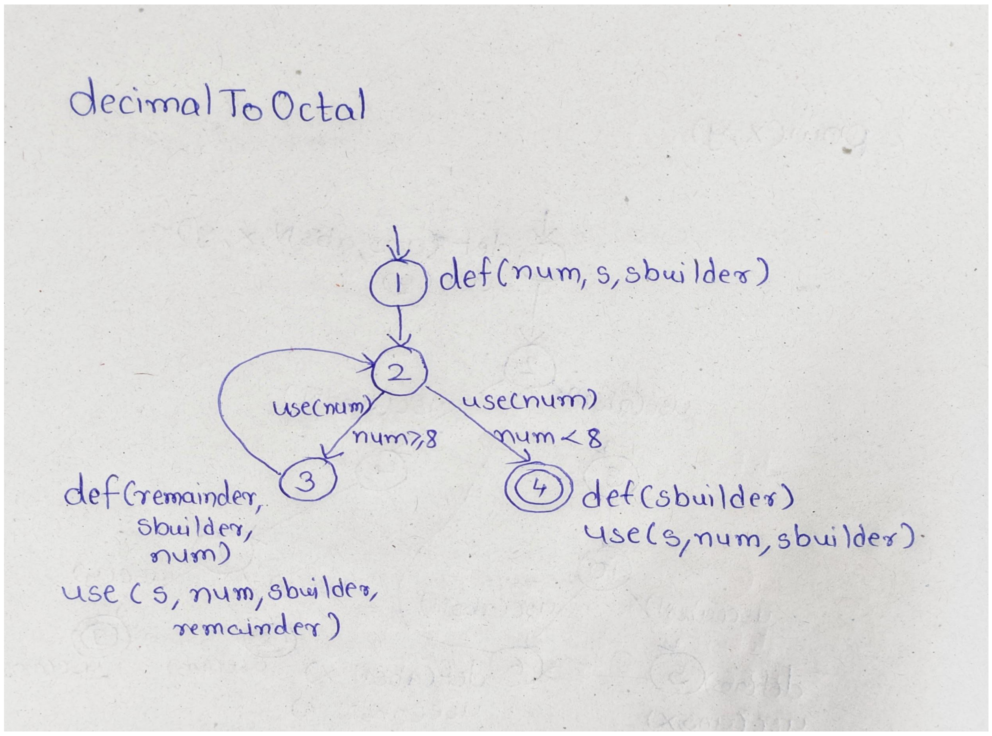
❖ This function converts decimal number into its octal equivalent.

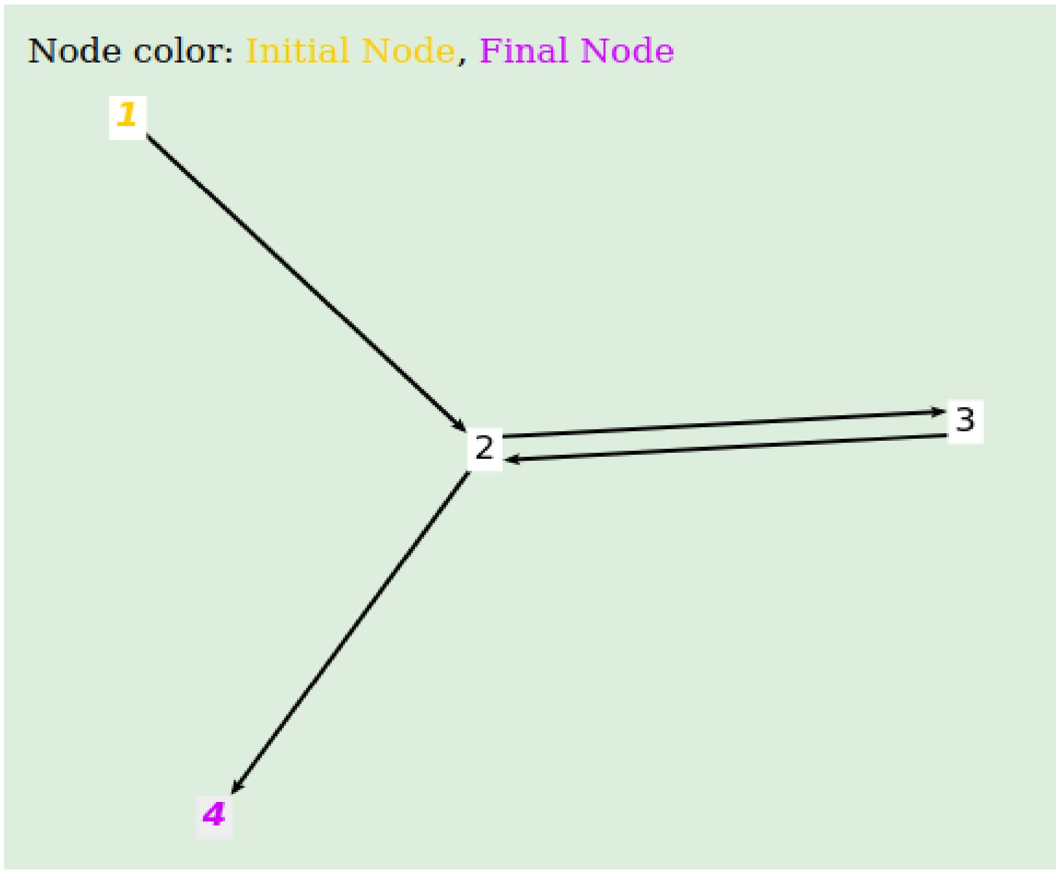
❖ Code:



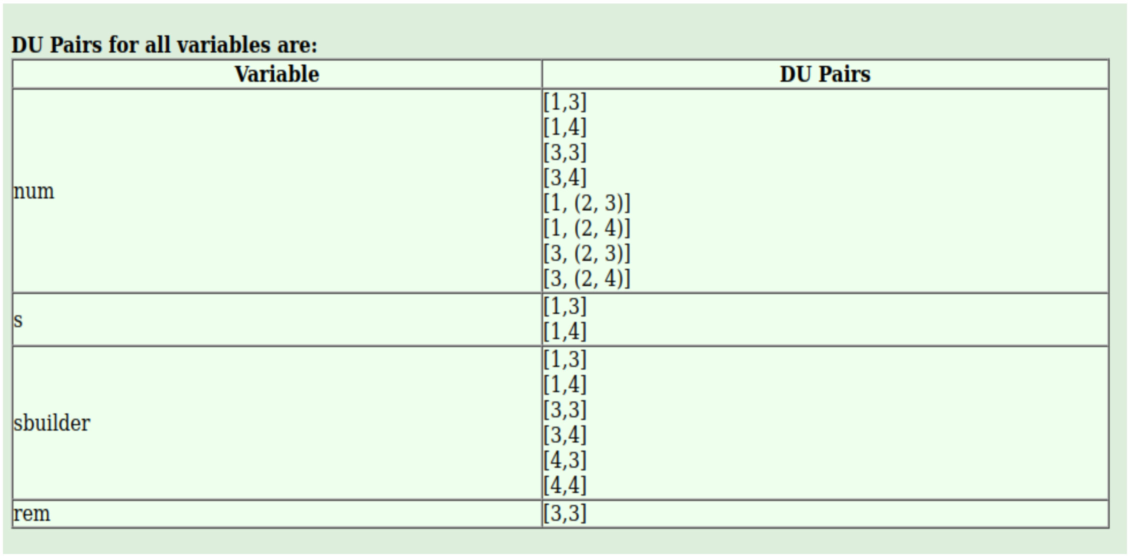
```java
//Decimal To Octal Conversion
public String decimalToOctal(int num) {
    String s = "01234567";
    StringBuilder sbuilder = new StringBuilder();
    while(num>=8)
    {
        int reminder = num%8;
        sbuilder.append(s.charAt(reminder));
        num=num/8;
    }
    sbuilder.append(s.charAt(num));
    sbuilder.reverse();
    return sbuilder.toString();
}
```

❖ Data Flow Graph:





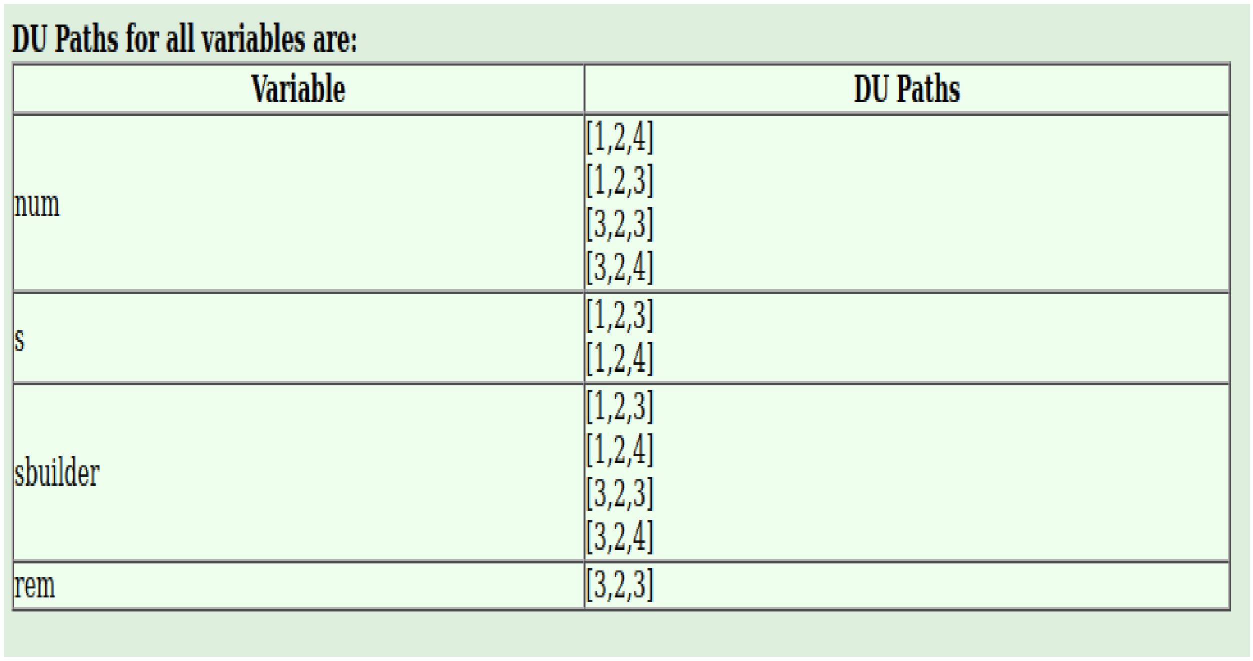
❖ DU pairs:



DU Pairs for all variables are:

Variable	DU Pairs
num	[1,3] [1,4] [3,3] [3,4] [1, (2, 3)] [1, (2, 4)] [3, (2, 3)] [3, (2, 4)]
s	[1,3] [1,4]
sbuilder	[1,3] [1,4] [3,3] [3,4] [4,3] [4,4]
rem	[3,3]

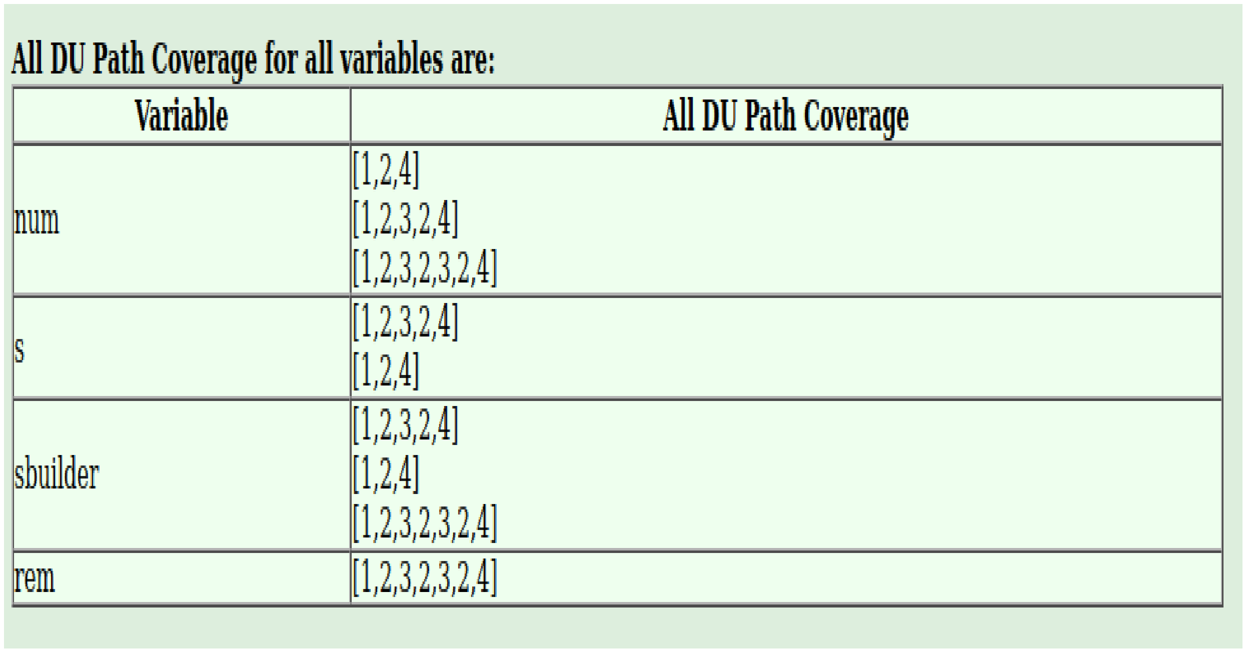
❖ DU paths:



DU Paths for all variables are:

Variable	DU Paths
num	[1,2,4] [1,2,3] [3,2,3] [3,2,4]
s	[1,2,3] [1,2,4]
sbuilder	[1,2,3] [1,2,4] [3,2,3] [3,2,4]
rem	[3,2,3]

❖ All DU path coverage



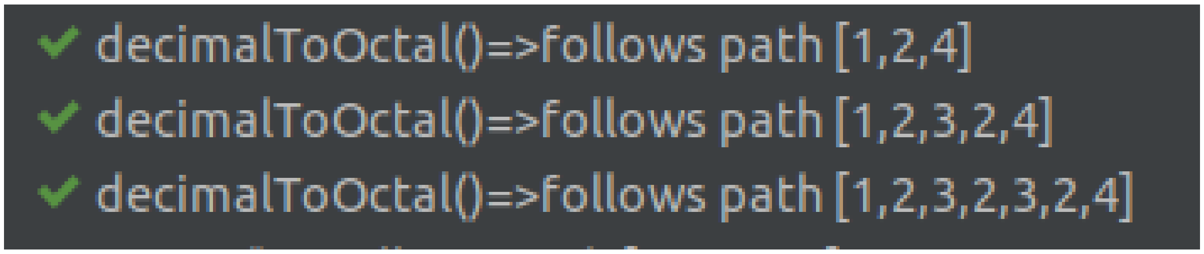
All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
num	[1,2,4] [1,2,3,2,4] [1,2,3,2,3,2,4]
s	[1,2,3,2,4] [1,2,4]
sbuilder	[1,2,3,2,4] [1,2,4] [1,2,3,2,3,2,4]
rem	[1,2,3,2,3,2,4]

There are total 3 unique paths:

1. That skips the loop: [1,2,4]
    - Test case for [1,2,4]:
        - decimalToOctal(num)⇒ decimalToOctal(7)
        - Expected output = 7
2. That requires at least one iteration of loop: [1,2,3,2,4]
    - Test case for [1,2,3,2,4]:
        - decimalToOctal(num)⇒ decimalToOctal(10)
        - Expected output = 12
3. That requires at least one iteration of loop: [1,2,3,2,3,2,4]
    - Test case for [1,2,3,2,3,2,4]:
        - decimalToOctal(num)⇒ decimalToOctal(164)
        - Expected output = 244

❖ Result:



```
✓ decimalToOctal()=>follows path [1,2,4]
✓ decimalToOctal()=>follows path [1,2,3,2,4]
✓ decimalToOctal()=>follows path [1,2,3,2,3,2,4]
```

Contributions

- Source Code, Data Flow Graphs, DU pairs, DU Paths, All DU Path Coverage test requirement and Test case design of pow() and addArrElements() is done by ***Abhinandan Jain (MT2021004)***.
- Source Code, Data Flow Graphs, DU pairs, DU Paths, All DU Path Coverage test requirement and Test case design of binaryToHexa() and integerToRoman() is done by ***Sagar Shirke (MT2021112).***
- Source Code, Data Flow Graphs, DU pairs, DU Paths, All DU Path Coverage test requirement and Test case design of fact() and decimalToOctal() is done by ***Naidu SKB (MT202137).***

References

- https://cs.gmu.edu:8443/offutt/coverage/DFGraphCoverage
- https://youtube.com/playlist?list=PLyqSpQzTE6M-sBjDcT21Gpnj8grR2fDgc
- https://ece.uwaterloo.ca/~agurfink/ece653/assets/pdf/W04-DataflowCoverage.pdf