Software Testing 1

Software Engineering
Prof. Maged Elaasar

Learning objectives

- Control flow test coverage
- Estimating the number of paths

Control Flow Test Coverage

How much testing is needed?

- Problem 1. Sometimes developers do not write enough tests.
- Problem 2. Sometimes they write too many redundant tests, causing overhead.
- Problem 3. During software evolution, we don’t have time to rerun all tests again. Identifying relevant tests to rerun is hard.

We need a way to tell how much testing is needed and where to focus!

Test Coverage

- One way of ensuring test adequacy is to increase code coverage by tests
- What tests can give us enough coverage for this code?
- We are going to focus on control flow test coverage which is based on the code's Control Flow Graph

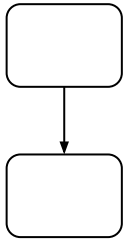
```
public class PathExample {

        public int returnInput (int x,
                boolean cond1,
                boolean cond2,
                boolean cond3) {

                if (cond1) {
                        x++;
                }
                if (cond2) {
                        x--;
                }
                if (cond3) {
                        x=x;
                }
                return x;
        }
}
```

Control Flow Test Coverage Criteria

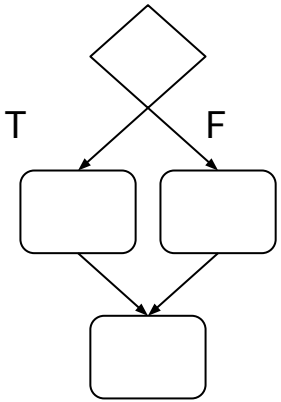
- Statement coverage
  - Percentage of statements exercised by tests
- Branch coverage
  - Percentage of branches (condition evaluations) exercised by tests
- Path coverage
  - Percentage of control flow paths exercised by tests

```
public class PathExample {

    public int returnInput (int x,
            boolean cond1,
            boolean cond2,
            boolean cond3) {

        if (cond1) {
            x++;
        }
        if (cond2) {
            x--;
        }
        if (cond3) {
            x=x;
        }
        return x;
    }
}
```

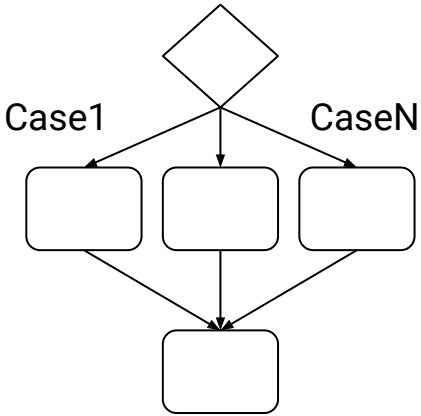
Control Flow Graph (CFG) Notation



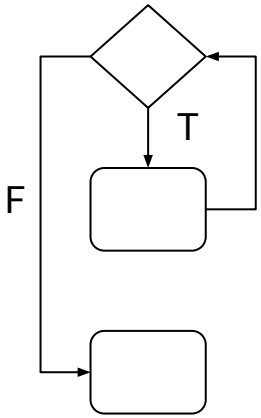
Sequence



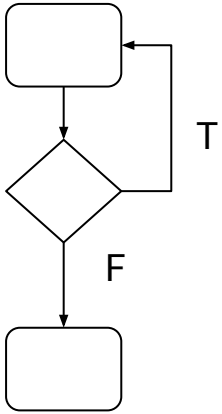
If-then-else



Switch



While



Do-While

Example 1

After execution of each statement in the test below, what is the **cumulative** statement, branch and path coverage respectively?

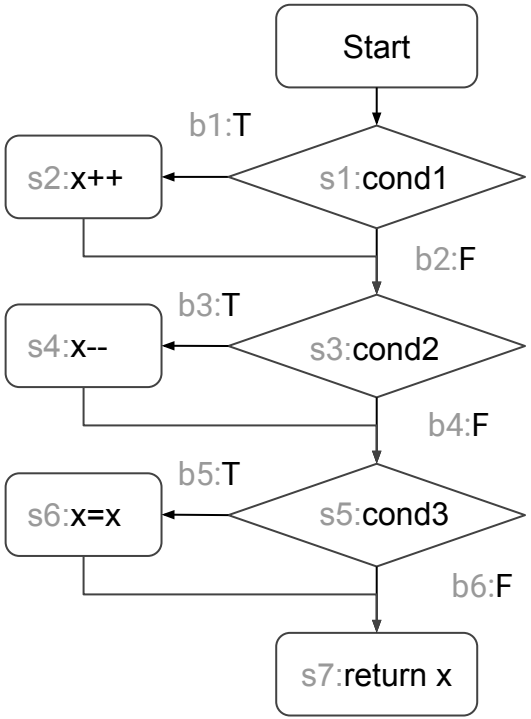
```
@Test public void testReturnInput () {
        PathExample p = new PathExample();
        assertEquals(p.returnInput(3, true, true, true), 3);
        assertEquals(p.returnInput(5, false, false, false), 5);
        assertEquals(p.returnInput(2, false, true, true), 1);
}
```

```
public class PathExample {

        public int returnInput (int x,
                boolean cond1,
                boolean cond2,
                boolean cond3) {

                if (cond1) {
                        x++;
                }
                if (cond2) {
                        x--;
                }
                if (cond3) {
                        x=x;
                }
                return x;
        }
}
```

Solution Step 1: Draw CFG

```
public class PathExample {

    public int returnInput (int x,
            boolean cond1,
            boolean cond2,
            boolean cond3) {

            if (cond1) {
                x++;
            }
            if (cond2) {
                x--;
            }
            if (cond3) {
                x=x;
            }
            return x;
    }
}
```

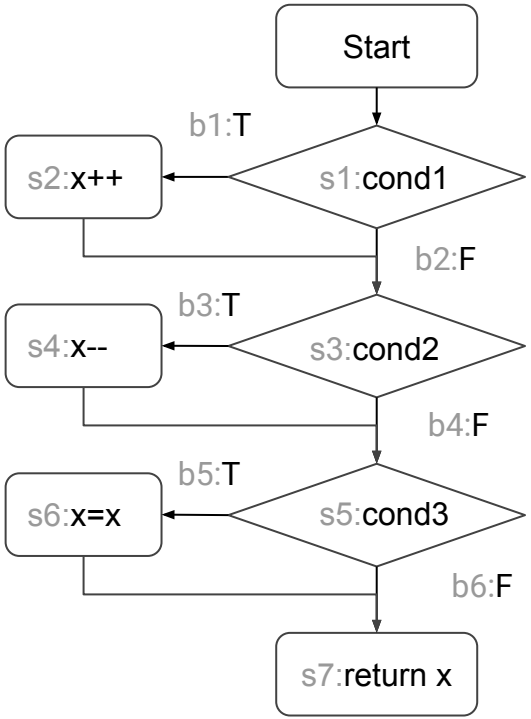
Draw the CFG of the code:



Solution Step 2: Fill Test Coverage Table

Fill out a code coverage table below by running the consecutive statements in the test (coverage is **cumulative**):

Input	Exercised Statements	Exercised Branches	Exercised Paths
(x=3, cond1=T, cond2=T, cond3=T)	s1, s2, s3, s4, s5, s6, s7	b1, b3, b5	[b1, b3, b5]
Coverage	100%	50%	12.5%
(x=5, cond1=F, cond2=F, cond3=F)	s1, s3, s5, s7	b2, b4, b6	[b2, b4, b6]
Coverage	100%	100%	25%
(x=2, cond1=F, cond2=T, cond3=T)	s1, s3, s4, s5, s6, s7	b2, b3, b5	[b2, b3, b5]
Coverage	100%	100%	37.5%



Branch Coverage

- Branch coverage is measured with respect to whether the decision takes the true or false branches.

- Suppose that we have a simple program
  if (x>1) x++ else x--;

- T1: x=2 makes the decision evaluate to T
- T2: x=0 makes the decision evaluate to F

- So when we have T1 only, it's 50% in terms of branch coverage, while when we have T1 and T2 we have 100%

- Is adding T3: x=3 necessary?

Example 2

- Consider a program with three decisions in a row.

```
public static int calculate(int x, int y, int z) {
        if (x>1) x++  else x--;
        if (y>2) y:=0 else y:=1;
        if (z>3) z:=0 else z:=2;
        return x+y+z;
}
```

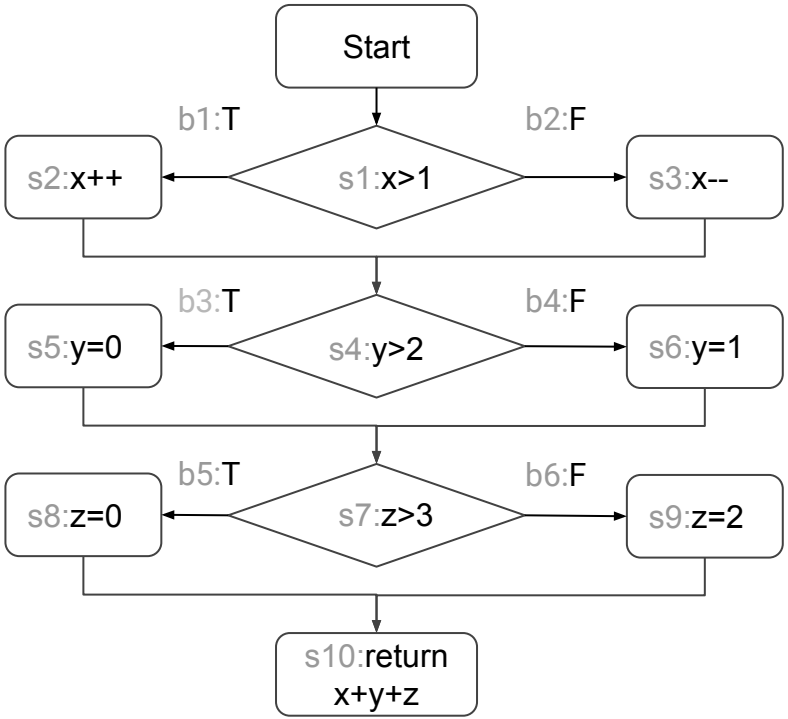
- What is a cumulative statement, branch and path coverage respectively after executing the following Test?

```
@Test
public void testReturnInput () {
        assertEquals(calculate(2, 3, 4), 3);
        assertEquals(calculate(0, 1, 2), 2);
}
```

Solution

```
public static int calculate(int x, int y, int z) {
        if (x>1) x++  else x--;
        if (y>2) y:=0 else y:=1;
        if (z>3) z:=0 else z:=2;
        return x+y+z;
}
```

Input	Exercised Statements	Exercised Branches	Exercised Paths
T1(x=2,y= 3,z= 4)	s1,s2,s4,s5,s7,s8 ,s10	b1,b3,b5	[b1, b3, b5]
Coverage	70%	50%	12.5%
T1(x=0,y= 1,z= 2)	s1,s3,s4,s6,s7,s9 ,s10	b2,b4,b6	[b2, b4, b6]
Coverage	100%	100%	25%



Estimating the Number of Paths

Estimating the number of paths

- Specifically, estimating the number of feasible loop iterations for bounded programs
- For a loop-free program with **k** decisions, the number of feasible paths is 2^**k**
  - Only consider decisions that are nondeterministic (i.e., may evaluate to **true** or **false**)
- How about a program with loops?
- In this case, we first perform loop **unrolling**, then the number of paths is 2^(# nondeterministic decisions)

Example 1

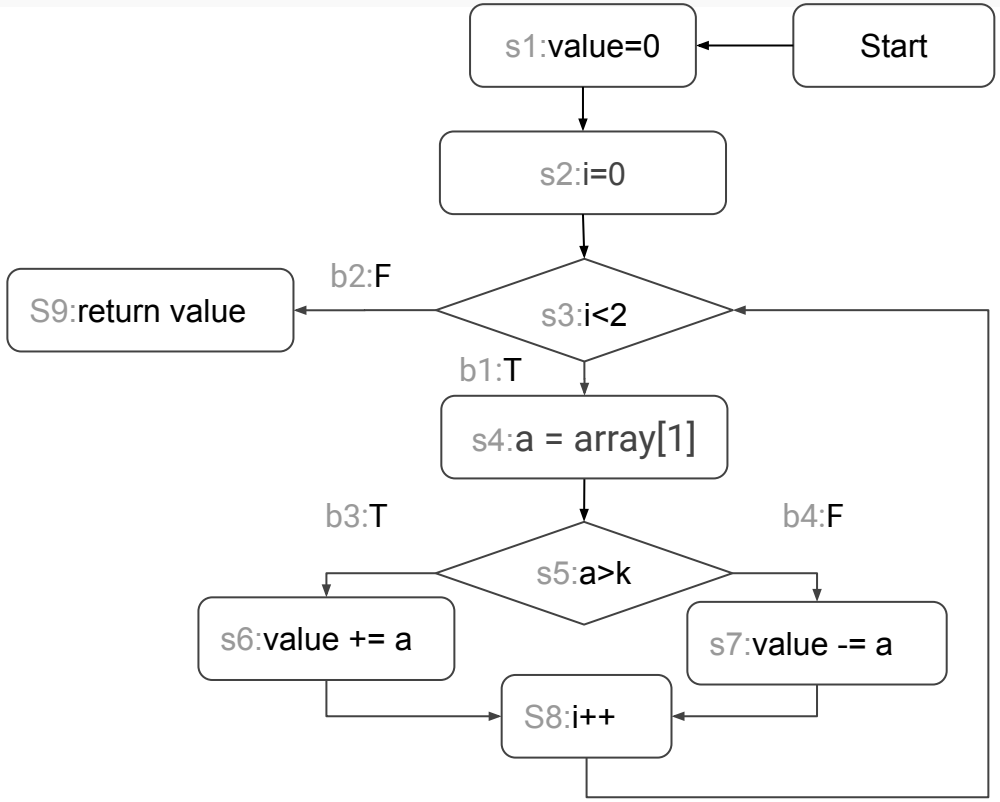
- What is the number of decisions in the loop-unrolled program?
- What is the maximum number of paths?
- After executing each statement in the test, what is the cumulative statement, branch and path coverage respectively?

```
public static int complexfun(int array[], int k) {
    int value = 0;
    for (int i=0; i<2; i++) {
        int a = array[i];
        If (a > k) {
            value = value+a;
        } else {
            value = value-a;
        }
    }
    return value;
}
```

```
@Test public void testComplexfun() {
    int a[] = {3, 5, 7};
    assertEquals(complexfun(a, 10), -8); // T1
    int b[] = {5, 6, 9, 11, 15};
    assertEquals(complexfun(b, 4), 11); // T2
    int c[] = {7, 2, 1, 2, 5, 6};
    assertEquals(complexfun(c, 4), 5); // T3
}
```

Draw CFG

```
public static int complexfun(int array[], int k) {
        int value = 0;
        for (int i=0; i<2; i++) {
                int a = array[i];
                If (a > k) {
                        value = value+a;
                } else {
                        value = value-a;
                }
        }
        return value;
}
```



Unroll Loop

```
public static int complexfun(int array[], int k) {
        int value = 0;
        for (int i=0; i<2; i++) { // bounded loop
                int a = array[i];
                If (a > k) {
                        value += a;
                } else {
                        value -= a;
                }
        }
        return value;
}
```

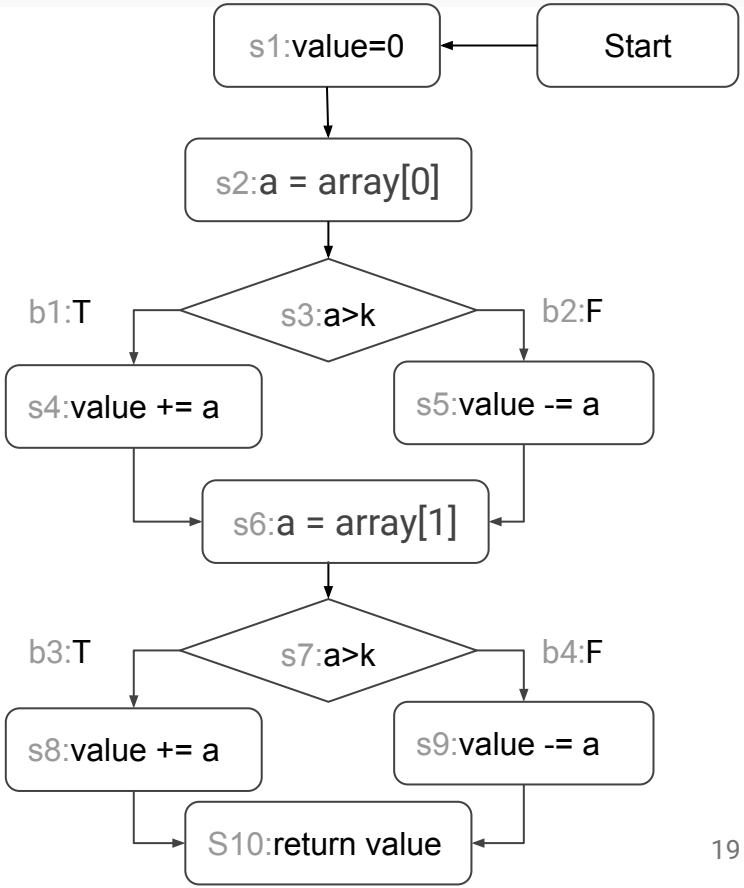
Loop Unrolling ==>

```
public static int complexfun(int array[], int k) {
        int value = 0;
        int a = array[0]; // first iteration
        If (a > k) {
                value += a;
        } else {
                value -= a;
        }
        a = array[1]; // second iteration
        If (a > k) {
                value += a;
        } else {
                value -= a;
        }
        return value;
}
```

Draw CFG of Loop Unrolled Version

```
public static int complexfun(int array[], int k) {
        int value = 0;
        int a = array[0]; // first iteration
        If (a > k) {
                value += a;
        } else {
                value -= a;
        }
        a = array[1]; // second iteration
        If (a > k) {
                value += a;
        } else {
                value -= a;
        }
        return value;
}
```

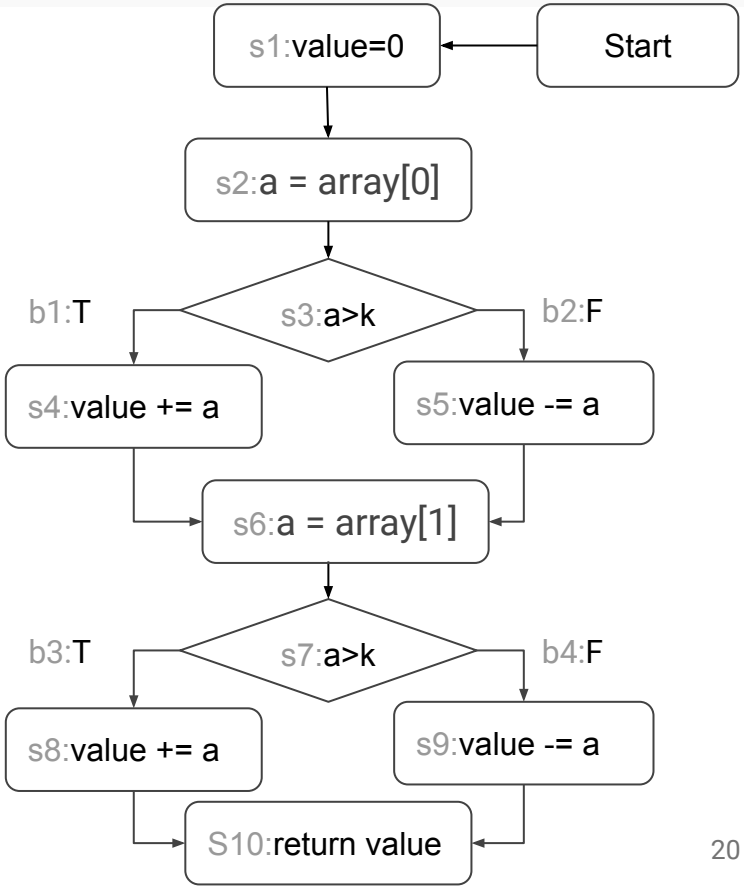
2 decisions that may take **T** or **F** ⇒ 2^2 = 4 paths



Calculate Coverage Criteria

```
@Test public void testComplexfun() {
        int a[] = {3, 5, 7};
        assertEquals(XXX.complexfun(a, 10), -8); // T1
        int b[] = {5, 6, 9, 11, 15};
        assertEquals(XXX.complexfun(b, 4), 11); // T2
        int c[] = {7, 2, 1, 2, 5, 6};
        assertEquals(XXX.complexfun(c, 4), 5); // T3
}
```

Input	Exercised Statements	Exercised Branches	Exercised Paths
T1	s1,s2,s3,s5,s6,s7,s9,s10	b2, b4	[b2,b4]
Coverage	80%	50%	25%
T2	s1,s2,s3,s4,s6,s7,s8,s10	b1, b3	[b1,b3]
Coverage	100%	100%	50%
T3	s1,s2,s3,s4,s6,s7,s9,s10	b1, b4	[b1,b4]
Coverage	100%	100%	75%



Example 2

- What is the number of decisions in the loop-unrolled program?
- What is the number of paths?

```
public static void fun2(int n) {
	int sum = 0;
	Random rand = new Random();
	for (int i=1; i<=n; i++) {
		for (int j=1; j<=Math.pow(3, i); j++) {
			if (rand.nextInt() % 2 == 0)
				sum++;
		 }
		for (int k=1; k<=i; k++) {
			if (Math.pow(2, k) % 2 == 0)
				sum++;
		}
	}
	System.out.println("n: "+n+"\tsum: "+sum);
}
```

About loop J's iteration

- When i is 1, the loop j executes 3^1 iterations
- When i is 2, the loop j executes 3^2 iterations
- ...
- When i is n, the loop j executes 3^n iterations

- T(n)=3^1 + 3^2 + ... + 3^n
  3T(n)= 3^2 + ... 3^n +3^(n+1)

- Subtract the first from the second
  2T(n) = 3T(n) - T(n) = 3^(n+1) - 3^1

- Divide the result by 2
  T(n) = (3^(n+1) - 3)/2 iterations

```
public static void fun2(int n) {
        int sum = 0;
        Random rand = new Random();
        for (int i=1; i<=n; i++) {
                for (int j=1; j<=Math.pow(3, i); j++) {
                        if (rand.nextInt() % 2 == 0)
                                sum++;
                 }
                for (int k=1; k<=i; k++) {
                        if (Math.pow(2, k) % 2 == 0)
                                sum++;
                }
        }
        System.out.println("n: "+n+"\tsum: "+sum);
}
```

About loop K's iteration

- When i is 1, the loop k executes 1 iteration
- When i is 2, the loop k executes 2 iterations
- ...
- When i is n, the loop k executes n iterations

- T(n)=1 + 2 + ... n // sum from 1 to n
- T(n) = n(n+1)/2 iterations

```
public static void fun2(int n) {
        int sum = 0;
        Random rand = new Random();
        for (int i=1; i<=n; i++) {
                for (int j=1; j<=Math.pow(3, i); j++) {
                        if (rand.nextInt() % 2 == 0)
                                sum++;
                 }
                for (int k=1; k<=i; k++) {
                        if (Math.pow(2, k) % 2 == 0)
                                sum++;
                }
        }
        System.out.println("n: "+n+"\tsum: "+sum);
}
```

Moreover

- Let’s consider the decisions within the j and k loops
- **j loop decision**: rand.nextInt()%2==0 can evaluate true or false, i.e., it is **nondeterministic**
- **k loop decision**: Math.pow(2,k)%2==0 is always true because (2^k)%2 is always 0, i.e., it is **deterministic** regardless of inputs provided.
- Total number of nondeterministic decisions is therefore (3^(n+1) - 3)/2
- Total number of paths is hence 2^{(3^(n+1) - 3)/2} paths

Extra point

- What if k loop decision was “rand.nextInt()%2==0” not “Math.pow(2,k)%2==0”?
- If so, the total number of iterations from loop J and K becomes: (3^(n+1) - 3)/2 **+** n(n+1)/2.
- The total number of paths is then 2^ {(3^(n+1) - 3)/2 + n(n+1)/2}

Example 3

- What is the number of decisions in the loop-unrolled program?
- What is the number of paths?

```
int sum = 0;
for (int i = 0; i < n; i++ ){
      for (int j = 1; j <= Math.pow(2,i); j++){
            if (a[i]<k)
               sum++;
      }
      for (int m=1; m<=i; m++) {
            if (pow(2,m) % 2 == 1)
                  sum++;
             else
                  sum--;
      }
}
```

Answer

- For the **m** loop, the decision (i.e., pow(2,m)%2) always evaluates to false for every possible input. So the decision is **deterministic**, hence does not contribute to increasing the number of paths.

- For the **j** loop
  When i=0, the inner j loop iterates 2^0
  When i=1, the inner j loop iterates 2^1
  ...
  When i=n-1, the inner j loop iterates 2^(n-1)

```
int sum = 0;
for (int i = 0; i < n; i++ ){
        for (int j = 1; j <= Math.pow(2,i); j++){
                if (a[i]<k)
                        sum++;
        }
        for (int m=1; m<=i; m++) {
                if (pow(2,m) % 2 == 1)
                        sum++;
                else
                        sum--;
        }
}
```

Answer (cont'd)

- T(n) = 2^0 + 2^1 + .. 2^(n-1)
- Multiply T(n) by 2 on both sides of the equation
- 2T(n) = 2^1 + .. + 2^(n-1) + 2^n
- Subtract the first equation from the second,
- T(n) = 2^n - 2^0
- T(n) = 2^n -1

- The decision (a[i]<k) could evaluate to true for some input and false for some other input, however it evaluates to the same value within each i.
- So the total number of paths is **2^n** not 2^(2^n-1).

Exercise (at home)

1. In example 3, assume that n=3. For a test suite with two cases T1=(k=2, a[]={4,3,5,4}) and T2=(k=1, a[]={2,7,5,10}). What is the cumulative path coverage of the test suite? Your answer must include the total number of feasible paths (in other words express it as a ratio: path/total-paths).

2. In example 3, assume that n=3. For a test suite with two cases T1=(k=2, a[]={2,3,1}) and T2=(k=3, a[]={2,3,1}). What is the cumulative path coverage of the test suite? Your answer must include the total number of feasible paths (in other words express it as a ratio: path/total-paths).

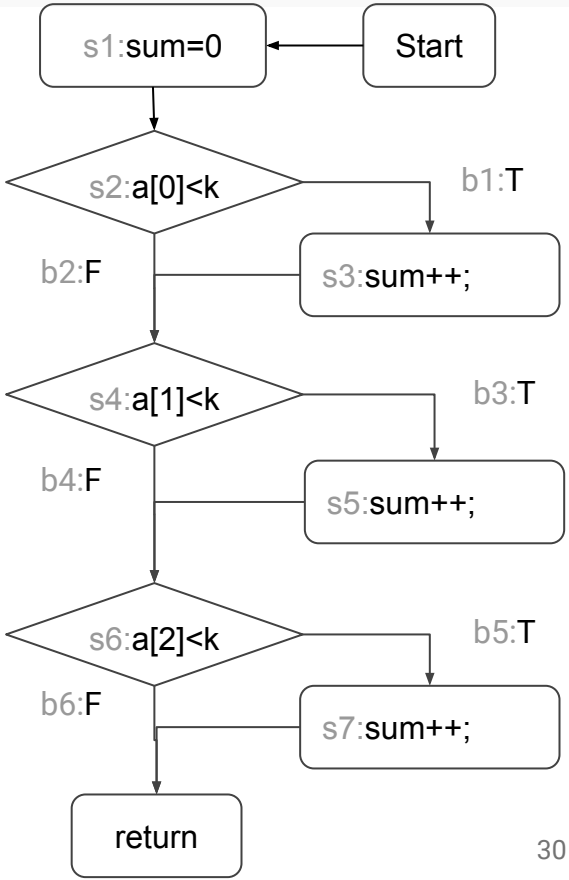
Exercise (at home) answer

1. The CFG on the right is the one for the function after loop unrolling and ignoring the deterministic conditions.

- T1=(k=2, a[]={4,3,5,4})
  paths: [b2, b4, b6]

- T2=(k=1, a[]={2,7,5,10})
  paths: [b2, b4, b6]

  Test suite cumulative path coverage = 1/8



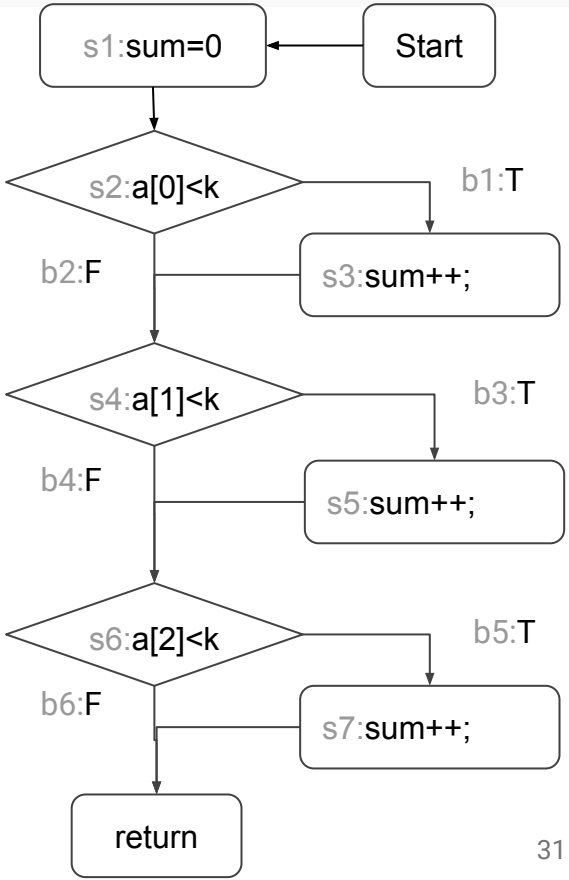
Exercise (at home) answer

2. The CFG on the right is the one for the function after loop unrolling and ignoring the deterministic conditions.

- T1=(k=2, a[]={2,3,1})
  paths: [b2, b4, b5]

- T2=(k=3, a[]={2,3,1})
  paths: [b1, b4, b5]

  Test suite cumulative path coverage = 2/8



Software Testing 2 Quiz

References

- Jorgensen, P.: “Software Testing, A Craftsman’s Approach,” CRC Press, 2013.
- Myers, G.: “The Art of Software testing,” John Wiley & Sons, 2004.
- Amman, P., Offutt, J.: “Introduction to Software Testing,” Cambridge, 2007.
- Buffoni, L.: “TDDD04: Software Testing”, Linkoping University, 2019.