**Homework 5:**

Question 1:

1. Attached to this Homework
2. 100% Node Coverage means that every node in the control graph is hit. This is my test case that would give 100% NC:

T = { t1 = {1, 2, 3, 5, 6, 7, 9, 8, 7, 9, 10, 11, 12, 13, 14, 15, 16, 4, 3, 17 } }

1. 100% Edge Coverage means that each edge in this control graph is hit. This is my test case that would give 100% EC:

T = { t1 = {1, 2, 3, 5, 6, 7, 12, 4, 3, 17},

t2 = {1, 2, 3, 5, 6, 7, 9, 8, 7, 9, 10, 11, 12, 13, 14, 15, 16, 4, 3, 17} }

1. (For this question, I will always refer to 100% EC because this also implies 100% NC; if each path is being taken, then it is for certain that each node will be touched).

In general, for most simple small to medium programs, it is possible to have 100% EC. This is when the program is coded with a procedural language, meaning that each line executes before the next. In object oriented programming, this also applies; each line inside a method/function will be called. It just will not be in a procedural order. We can argue that there is a possibility of not obtaining 100% EC if dead code lives inside the program. For example, if there is a method in a class that never gets called, then that method will never run at runtime, classifying it as dead code. This can also occur in procedural programming, but it is less frequent. The main argument that I will give that disagrees with the possibility of obtaining 100% EC is when a program uses threading or implements timing functions. There can be times when a block of code should be called logically, but will never run because of a timing/threading issue; if a chunk of code must run 3ms after a certain function, but that function always returns after 5ms, that specific chunk of code will never run. This means that there will never be an instance of time where this path gets taken in edge coverage, hence not getting 100% EC. This can also occur with Network programming, or any type of programming where timing or threading is used. However, for general purposes, it should be possible to get 100% EC.

Question 2:

1. Attached to this Homework
2. If we call C1.D, in theory, it will be downcasted first and search if C2 has D (since D does not exist in class C1. If C2.D exists, it calls C2.D. If C2.D does not exist, it will keep downcasting until it finds a child class with D (C\_.D). This implementation is illegal in Java because downcasting is not supported in the compiler (upcasting is though). However, this can be done with other languages such as C++ using pointers.

Question 3:

1. Line 6: if(i < 1)

If we change line 6 to the line above, we remove the condition where I == 1. Normally, if I == 1, then it passes through the condition. However, this mutation does not allow this behavior. Therefore, a test case that will kill this mutant is T = { t1 = { 1 } }. In the mutated program, the return value is 0 because fib = 0 in line 4. Nonetheless, the original program has of output 1, because the value of fib changes in line 7, when the if statement is true.

1. Line 6: if(i == 1)

If we have i == 0, the original program should output fib = 1 (line 7, then line 14). However, when i == 0 in the mutated program, it does not go through the if conditional statement, thus keeping the value of fib = 0 (line 4). This test case, where i == 0, will kill the mutant. This will also work with any negative number. In the original program, any negative input i will satisfy the first conditional statement, making fib = 1. However, the mutant will output fib = 0 (like the previous case), thus killing the mutant. Here is the test case that will kill this mutant: T = { t1 = { for all i < 1 } }

1. Line 12: fib2 = fib;

In the original program, any i > 1 will go through the else clause, hence passing by this statement. We can see that the mutant will output a different result than the original program. For example, if we pass in i with value 3, it should give us a result of 5. However, the mutant will output an 8. Essentially, the mutant will always output a 2­­i, so in this case 23 = 8. If we dig a little bit deeper though, and pass i in as 2, we end up with both programs, original and mutant, outputting the same result of 2. This is bad because it does not kill the mutant (same results), thus confirming that our dataset is not good enough. Taking this into consideration, this should be the test case for this mutant:

T = { t1 = { for all i > 2 } }.