林肯 FIT5171 保过班资料

Exam

内部资料 禁止外传

By 林肯 5171 教研组

Question 1 .... **15 marks**
Consider a program *FizzPrime* that takes as input two non-negative integers, x and i, both between 0 and 100, both inclusive. The number x is a prime numbers. As output, the program prints the number i itself within the range ([0, 100]) when it is not divisible by x. For multiples of x, but not multiples of x^2, the program should print "`Fizz`" instead of the number. For multiples of x^2 but not multiples of x^3, the program should print "`Prime`". Finally, for numbers which are multiples of x^3 the program should print "`FizzPrime`" instead.

(a) (5 marks) Develop robust equivalence classes for the input variables x and i given the above specification.

x

invalid:1. X < 0 ; 2 X >100;3, x is not a prime number

valid: x >= 0 ,x<=100,x is a prime number

i

invalid:1. i < 0 ; 2 i >100;

valid: i >= 0 ,i<=100

Equivalence (i, Fizz, Prime, FizzPrime)

R1 = {x,i is valid, i cannot be divided by x}

R2 = {x,i is valid, i canbe divided by x but cannot be divided by x^2}

R3 = {x,i is valid, i canbe divided by x^2 but cannot be divided by x^3}

R4 = {x,i is valid, i canbe divided by x^3}

R5 = {x <0 ,i is valid} R6 = {x >100, i is valid} R7 = {X is not a prime, i is valid}

R8 = {i <0 ,x is valid} R9 = {i >100, x is valid}

Test cases	x	i	Expected Output
R1	3	4	4
R2	3	6	Fizz
R3	3	9	Prime
R4	3	27	FizzPrime
R5	−1		Error input
R6	101		Error input
R7	4		Error input

(b) (6 marks) Develop test cases using the *robust* (not worst-case) version of the boundary value testing technique.

Test cases	x	i	Expected output
1	−1	7	Invalid input
2	0	7	Invalid input
3	1	7	Invalid input
4	7	7	Fizz
5	99	7	Invalid input
6	100	7	Invalid input
7	101	7	Invalid input
8	7	−1	Invalid input
9	7	0	0
10	7	1	1
12	7	99	99
13	7	100	100
14	7	101	Invalid input

(c) (4 marks) You have been given the task of performing blackbox testing on an implementation of the above algorithm. Of the main blackbox testing techniques we have discussed: boundary value testing (BVT), special value testing (SVT), equivalence class testing (ECT), and decision table-based testing (DTT), explain why each technique is (or is not) appropriate.

BVT: Not a appropriate, because not consider the x is a prime and output logic.

SVT: is appropriate, because consider the x is a prime and output logic.

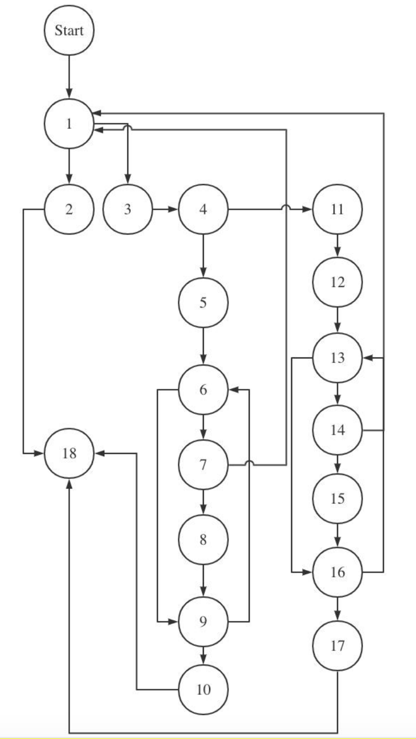
ECT: is appropriate

DTT: is appropriate, it consider the out put logic

$minimax(node, depth, maximisingPlayer)$

```
1  if depth = 0 ∨ is_terminal(node) then
2  |   return the heuristic value of node
3  end
4  if maximisingPlayer then
5  |   bestValue ← −∞
6  |   foreach child of node do
7  |   |   val ← minimax(child, depth − 1, false)
8  |   |   bestValue ← max(bestValue, val)
9  |   end
10 |   return bestValue
11 else
12 |   bestValue ← +∞
13 |   foreach child of node do
14 |   |   val ← minimax(child, depth − 1, true)
15 |   |   bestValue ← min(bestValue, val)
16 |   end
17 |   return bestValue
18 end
```

(a) (5 marks) Draw the program graph for the above function.



(b) (1 mark) Calculate the cyclomatic complexity of the program graph in the previous part.

C= E−V+2p

Question 3 . . . . **6 marks**

One of the goals of integration testing is to be able to isolate faults when a test case causes a failure. Consider integration testing for a program written in a procedural/object-oriented programming language. Rate the following integration strategies on their abilities of (1) relative fault isolation and (2) testing of co-functionality.

You also need to provide a *rationale* for your answer.

A Big bang

B Decomposition-based top-down integration

C Decomposition-based bottom-up integration

D Decomposition-based sandwich integration

E Call graph-based pairwise integration

F Call graph-based neighbourhood integration (radius 1)

G Call graph-based neighbourhood integration (radius 2)

Show your ratings graphically by placing the letters corresponding to a strategy on a line, as in the example below. Suppose that for the ability of fault isolation, strategies X and Y are about equal and not very effective, and strategy Z is very effective.

Note that this rating is relative and qualitative, so don't agonise over where *exactly* to put a strategy, but focus on their *relative* position.

Fault isolation

Y
X — Z
Low — High

A G F E D B
C

Question 4 . . . . **8 marks**

In lecture 8 we introduced Weyuker's 9 properties to evaluate software metrics. Some of the properties (for example, properties 1, 3, 4 and 8) are quite simple and intuitive. However, some other properties are a bit more complex and need further analysis.

Weyuker's property 9 states that the complexity of the composition of two programs may be greater than the sum of the complexities of the two taken separately. More formally,

$$\exists A, B : Program \bullet M(A) + M(B) < M(A+B)$$

where M represents a given metric and $A+B$ represents the composition of A and B.

The structural metric *depth of nesting* of a program P, denoted $n(P)$, is defined for programs that only contain *structured programming constructs.*

Given a program, the repeated application of the following two operations can be used to decompose it into a unique tree of structured programming constructs.

Sequence: composing two program graphs sequentially by merging one program graph's terminal node with the other program graph's initial node. For example, sequential composition of programs A and B is denoted by $A; B$.

Sequence: composing two program graphs sequentially by merging one program graph's terminal node with the other program graph's initial node. For example, sequential composition of programs A and B is denoted by $A; B$.

Nesting: replacing one node in one program with the entirety of another program. For example, nesting program B in program A at node x of A is denoted by $A(B, x)$.

The depth of nesting values of programs constructed by the above two operations are defined as below.

Sequence: $n(P_1; P_2; \ldots; P_n) = \max(n(P_1), n(P_2), \ldots, n(P_n))$, and

Nesting: $n(P_1(P_2; \ldots; P_n)) = 1 + \max(n(P_2), \ldots, n(P_n))$, where $P_2, \ldots, P_n$ are sequentially nested inside P_1.

Recall that there are six basic types of structured programming constructs:

Construct	Description	Construct	Description
P_n	sequence $(n = 1, 2, \ldots)$	D_2	while loop
D_0	if-then	D_3	do-while loop
D_1	if-then-else	C_n	case-switch

The depth of nesting value for all the above constructs is 1 except for P_1, which is 0. The depth of nesting value of a program is calculated in a bottom-up fashion.

For Weyuker's property 9 and the metric depth of nesting $n(P)$ of a valid program P, do the following:

(a) State whether the property holds or not.

(b) Prove your claim (informally).

Sequence：

n(A) = n(px)

n(B) = n(py)

n(A+B) = n(px;py) =max(n(px),n(py))

n(A) or n(B) 的最大值是 n(A+B) 的值，所以 n(A+B) == n(A) or n(b)，推导出 n(A+B) <= n(A) + n(b)，所以不等式不成立

Nesting：

n(A) = n(p1) = 0

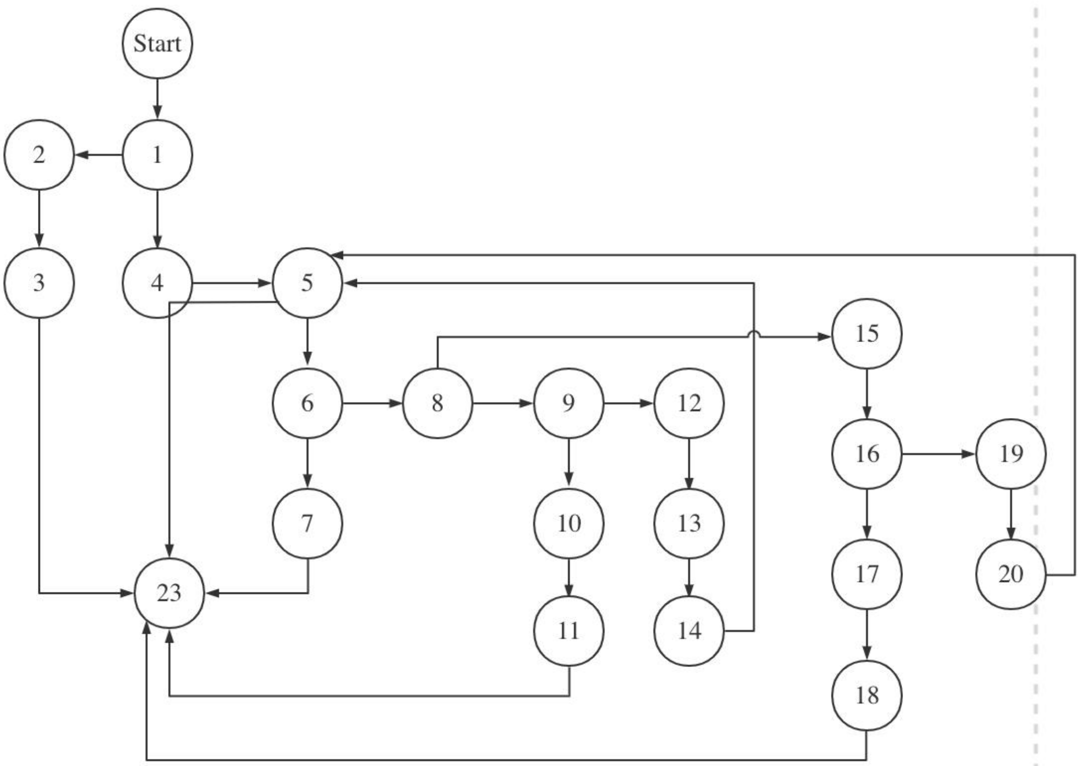
n(B) = n(p2) = 1

n(A+B) = n(p1(p2)) = 1 + max(n(p2)) = 2

n(A) + n(B) < n(A+B)

不等式成立

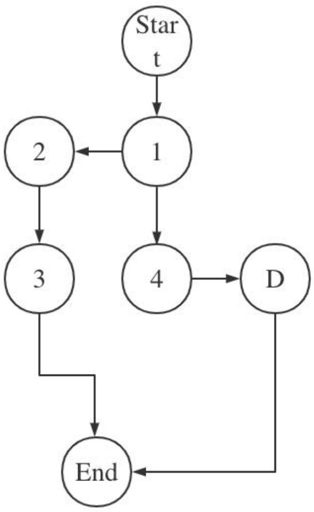
```
Input: node                                        ▷ Node to be inserted.
Input: root                                        ▷ The root node of the BST.
if root = null then
    root ← node
    return
end
while root ≠ null do
    if node = root then                            ▷ Node already in the BST
        return
    else if node < root then                       ▷ Insert left
        if root.left = null then
            root.left ← node
            return
        else
            root ← root.left
        end
    else                                           ▷ Insert right
        if root.right = null then
            root.right ← node
            return
        else
            root ← root.right
        end
    end
end
```



(b) (5 marks) Recall that McCabe's essential complexity measures how *unstructured* the logic of a program is by calculating the Cyclomatic complexity of the condensed program graph. In this part,

i. draw the final condensed graph for the program graph you came up with in part (a) above, and

ii. calculate the Cyclomatic complexity of the condensed graph you draw.



Question 5 .... **15 marks**

Mutation testing is a technique to assess the efficacy and quality of a test suite. It works by making *mutants*, syntactic variations of the program under test, and measures how many of the mutants are *killed* by the test suite. The presence of non-equivalent *live* mutants represents inadequacy of the test suite.

The following Java method, `min`, returns the smallest of three integer parameters.

```
1  public int min(int a, int b, int c) {
2      int temp = a;
3      if (b < a) {
4          temp = b;
5      }
6      if (c < b) {
7          temp = c;
8      }
9      return temp;
10 }
```

(a) (3 marks) Come up with an *equivalent* mutant by applying a *first-order* mutation. In your answer, identify:

1. The mutation operator applied,
2. The associated statement to be changed, and
3. What the statement is changed to.

Conditionals Boundary Mutator 或者 ROR

line3 change to b<=a

(b) (8 marks) Devise a set of three test cases that achieves 100% statement coverage. Come up with three *non-equivalent* first-order mutants of the original program, making use one of the following mutation operators in each mutant. Determine the *kill rate* of your test suite on the three mutants.

The mutation operators you can use are:

`ror` Relational operator replacement.

`sdl` Statement deletion.

`uoi` Unary operator insertion.

Test cases	a	b	c	Expect output
1	3	2	1	1
2	1	2	3	1
3	2	3	1	1

ROR	line 3:b > a	Kill tc1
SDL	remove line 4	Kill tc3
UOI	line2:a++	kill tc1

All MT be killed by test cases, so the rate is 100%.

(c) (4 marks) Is there a defect in the program? If so, develop the smallest set of test cases that achieves 100% statement coverage but *does not* reveal the defect. If not, develop the smallest set of test cases that achieves 100% statement coverage.

There is a defect, when a=1, b=3,c=2, output is 2. It is not a min number.

Test cases	a	b	c	Expect output
1	3	2	1	1
2	1	2	3	1