1st Sem. 2021-22
Software Testing
End Sem (Closed Book)

==========

Time: 2.5 hours Marks: 60

==========

I. Assert T if the assertions below are true, otherwise mark F. Justify your answer **[10M].**

a. Bottom-up testing is not an effective way to reveal architectural design flaws.
True. *It may reveal such flaws, but isn't an effective way to do so. Topdown testing is an effective way to reveal architectural design flaws.*

b. One drawback of bottom-up system testing (as compared to top-down system testing) is that when a test fails, there are more places you have to look for the bug.
True. *Unless unit testing was perfect, the bug might appear in any class that was executed as part of the test. In top-down testing, all the system tests are written at the beginning and are simply re-run with fewer and fewer stubs.*

c. A refactoring is generally motivated by some specific task it will make easier, as opposed to simply improving the code.
True*. It's not cost-effective to refactor just to reduce entropy or satisfy your aesthetics. When a specific task (fix bug #2242, or add a particular feature) would be simplified if the code had a different structure, then it makes sense to first complete the refactoring, and after that to do the task, rather than to mix together the code restructuring and the task.*

d. Including the users/customers in the process of designing the architecture will yield a better product
True.

e. It is theoretically possible for testing to prove the absence of bugs.
True. *Two ways testing can demonstrate the absence of bugs are Exhaustive testing and testing using revealing subdomains. However, such approaches are usually impractical.*

f. Boundary testing can catch off-by-one errors, but not null pointer exceptions.
False. Boundary testing can catch off-by-one errors, but not null pointer exceptions

g. When designing tests, if partitions are chosen perfectly, there is no point to testing boundary values near the edges of the partition.
True. *Suppose your partitions are chosen perfectly. If the program behaves properly for any element in the whole partition, then it behaves properly for every element in the whole partition, including ones at the boundary. You don't have to test multiple boundary values — just test any one value.*

h. The number of modules in a system is one way to measure complexity of that system.
True.

i. Visiting all the states that a program has assures that you've also traversed all the transitions among them
False

j. Testing is just another name for debugging.

False

2. Circle all that apply. Note you need to marks only and all correct answers.**[2M]**

(a) Black-box tests generally aim to provide as much coverage of the lines of code in a module as possible, because coverage is usually positively-correlated with test quality.
(b) Black-box tests can be used to find aliasing errors that occur when two different formal parameters of a method both refer to the same object.
(c) Black-box tests can be effective at detecting representation exposure.
(d) In practice, it is often difficult to re-use the same black-box test when crucial parts of the implementation change, even when the specification remains the same.
(e) Black-box tests are able to partition the input space into finer (smaller) partitions than glass-box tests.

Ans. b, c

3. Circle all that apply. Note you need to marks only and all correct answers. **[2M]**

Consider the following statements about maintenance testing:
(a) It requires both re-test and regression test and may require additional new tests.
(b) It is testing to show how easy it will be to maintain the system.
(c) It is difficult to scope and therefore needs careful risk and impact analysis
(d) It need not be done for emergency bug fixes.

Ans. a,c

4. Which of the following on extreme programming are true. Note you need to marks only and all correct answers**.[2M]**

a. Because of pair programming, it requires twice the number of developers.
b. In XP, code is rarely changed after written
c. XP is a interactive process development process
d. Customer does not need to produce any requirement in XP
e. XP follows the test-driven development

Ans. c,e

5. Consider the following code. Compute the operands, operators and program estimated length **[4M].**

```
#include main() {

int a ; scanf (“%d”, &a);

if ( a >= 10 )

if ( a < 20 )

printf ("10 < a< 20 %d\n" , a);

else printf ("a >= 20 %d\n" , a);

else printf ("a <= 10 %d\n" , a);

 }
```

[Be lenient in marking here]

Operators	Number of occurrences	Operators	Number of occurrences
#	1	<=	1
include	1	\n	3
stdio.h	1	printf	3
< … >	1	<	3
main	1	>=	2
(…)	7	if … else	2
{ … }	1	&	1
int	1	,	4
;	5	%d	4
scanf	1	“ … “	4
$\mu_1 = 20$		$N_1 = 47$	

Note header file is missing in the question, so mark accordingly. If the answer is 19, still give the marks. µ2 = 3

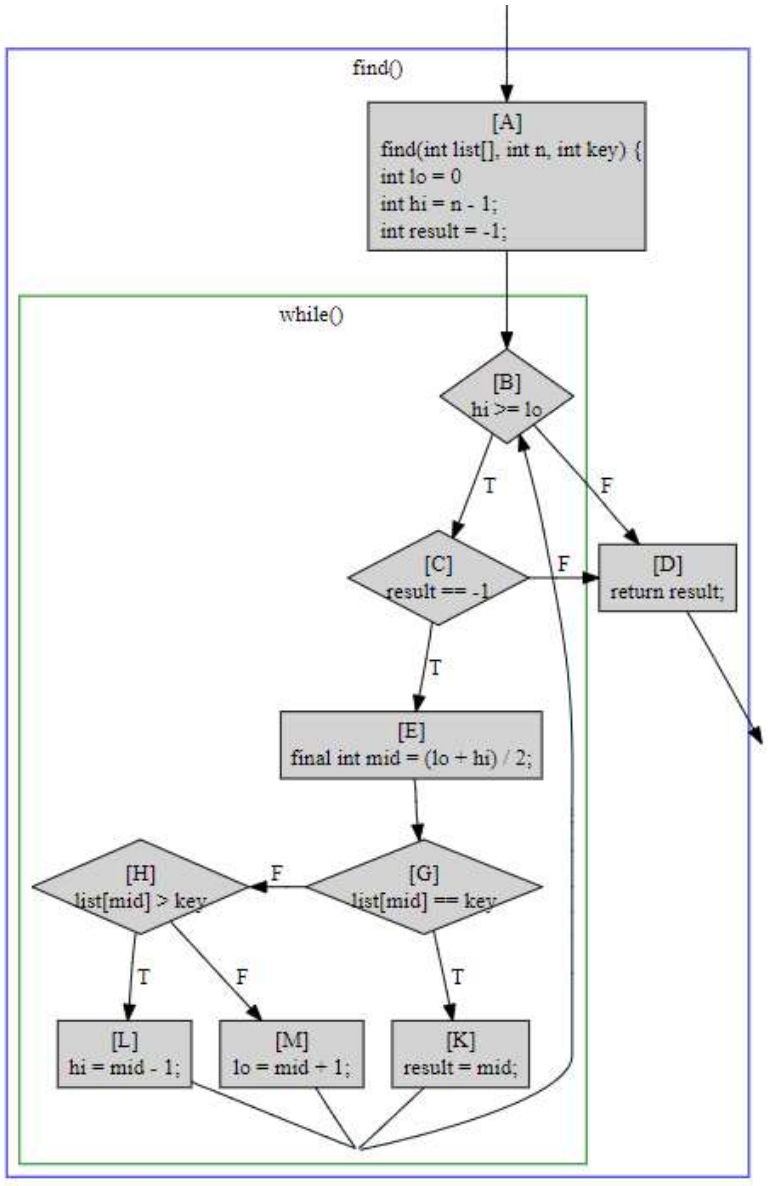
Program Estimated length:

$$\hat{N} = \mu_1 \log_2 \mu_1 + \mu_2 \log_2 \mu_2$$

3. Consider the following CFG for involving the method find() that implements a binary search on an ordered list of integers, in order to find the index of any element with value *key*. For example, on these inputs:

list[] = { 0, 3, 5, 8, 8, 17 }
n = 6
key = 5

find() should return 2 (the index at which we find the key value 5).



Consider the following test case: list[] = { 0, 3, 5, 8, 8, 17 }, n = 6, key=8

a. Does it satisfy the the “all DU pairs” criterion? **[5M]**

No.

To cover “all DU pairs”, the test suite must include a def-clear path from each variable definition to all uses of that variable that are reachable from the definition.

```
list[] = { 0, 3, 5, 8, 8, 17 }
  n = 6
  key = 8
```

Executing this, we start off on the first iteration with *lo* = 0 and *hi* = 5. *mid* will be 2 (since Java rounds ints down), and since *list[2]* = 5 (< *key*), we'll execute node M and start the next iteration with *lo* = 3 and *hi* = 5. *mid* will now be 4, and since *list[4]* = 8 (== *key*), we'll set *result* in node K, and execution will terminate via node C on the next iteration. So our execution path will have been as follows:

```
A BCEGHM BCEGK BCD
```

We can then cross off DU pairs covered in our table. Note that because *lo*, *hi* and *result* can be defined in more than one place, we have to be careful that the paths we're looking at are def-clear paths, e.g. for *result*'s A→C path, there must be no instance of node K redefining *result* between A and C.

Variable	Definitions	Def-use pairs
list[]	A	~~A→G~~, ~~A→H~~
n	A	~~A→A~~
key	A	~~A→G~~, ~~A→H~~
mid	E	~~E→K~~, E→L, ~~E→M~~, ~~E→G~~, ~~E→H~~
lo	A, M	~~A→E~~, ~~A→B~~, ~~M→E~~, ~~M→B~~
hi	A, L	~~A→E~~, ~~A→B~~, L→E, L→B
result	A, K	A→D, ~~A→C~~, ~~K→D~~, ~~K→C~~

Looking at the left-over DU pairs here, we need to exercise the DU pairs involving L and to also exercise the AD pair which can only occur when the list is empty.

b. Provide the test suites for to satisfy all DU pairs[**10M**].

	list[]	n	key
Test 1	{ 0, 3, 5, 8, 8, 17 }	6	8
Test 2	{ 0, 3, 5, 8, 8, 17 }	6	3
Test 3	{ }	0	1

4. Given the following code, following test cases are designed as a table. For each test suite comment on the coverage criterion: Statement coverage, Branch coverage, MC/DC coverage, All-uses coverage**[10M]**

```
public int Segment(int t[], int l, int u){

   // Assumes t is in ascending order, and l<u,
   // counts the length of the segment
   // of t with each element l<t[i]<u

   int k = 0;

   for(int i=0; i<t.length && t[i]<u; i++){

      if(t[i]>l){k++;}

   }
   return(k);
}
```

Test suite	Test number	t[]	l	u	Expected
1	1	{}	1	10	0
	2	{0}	0	0	0
	3	{0}	-1	1	1
2	1	{3, 4, 5, 6}	4	8	2
	2	{}	4	8	0
	3	{9}	9	10	0

Solution

1 →4

2→5

First column is of statement coverage, and following: Branch coverage, MC/DC coverage, All-uses

4	1	{}	1	10	0	✓	×	×	×
	2	{0}	0	0	0				
	3	{0}	-1	1	1				
5	1	{3, 4, 5, 6}	4	8	2	✓	✓	×	✓
	2	{}	4	8	0				
	3	{9}	9	10	0				

Let the following two mutants be introduced in the code:

M1: for(int i = 0; i > t.length && t[i] < u; i++) and

Yes for 1 and 2

M2: if(t[i] < l)

Yes for 1 and 2

From the test suites given in the table, identify the test suite(s) that kills the aforementioned mutants. **[5M]**

5. Consider a program **P**, which reads three numbers 'a', b and 'c', and prints the highest among these numbers. Valid test cases for this program **P** are also shown in table below

Program P

1. int main(){
2. int a,b,c;
3. input(a,b,c);
4. if a > b
5. if a > c
6. print 'a' as highest
7. else
8. print 'c' as highest
9. endif
10. else
11. if b > c
12. print 'b' as highest
13. else
14. print 'c' as highest

Valid Test Cases for P

Test case	a	b	c
T1	10	6	8
T2	6	10	8
T3	6	8	10
T4	5	5	3
T5	3	5	5
T6	5	3	5

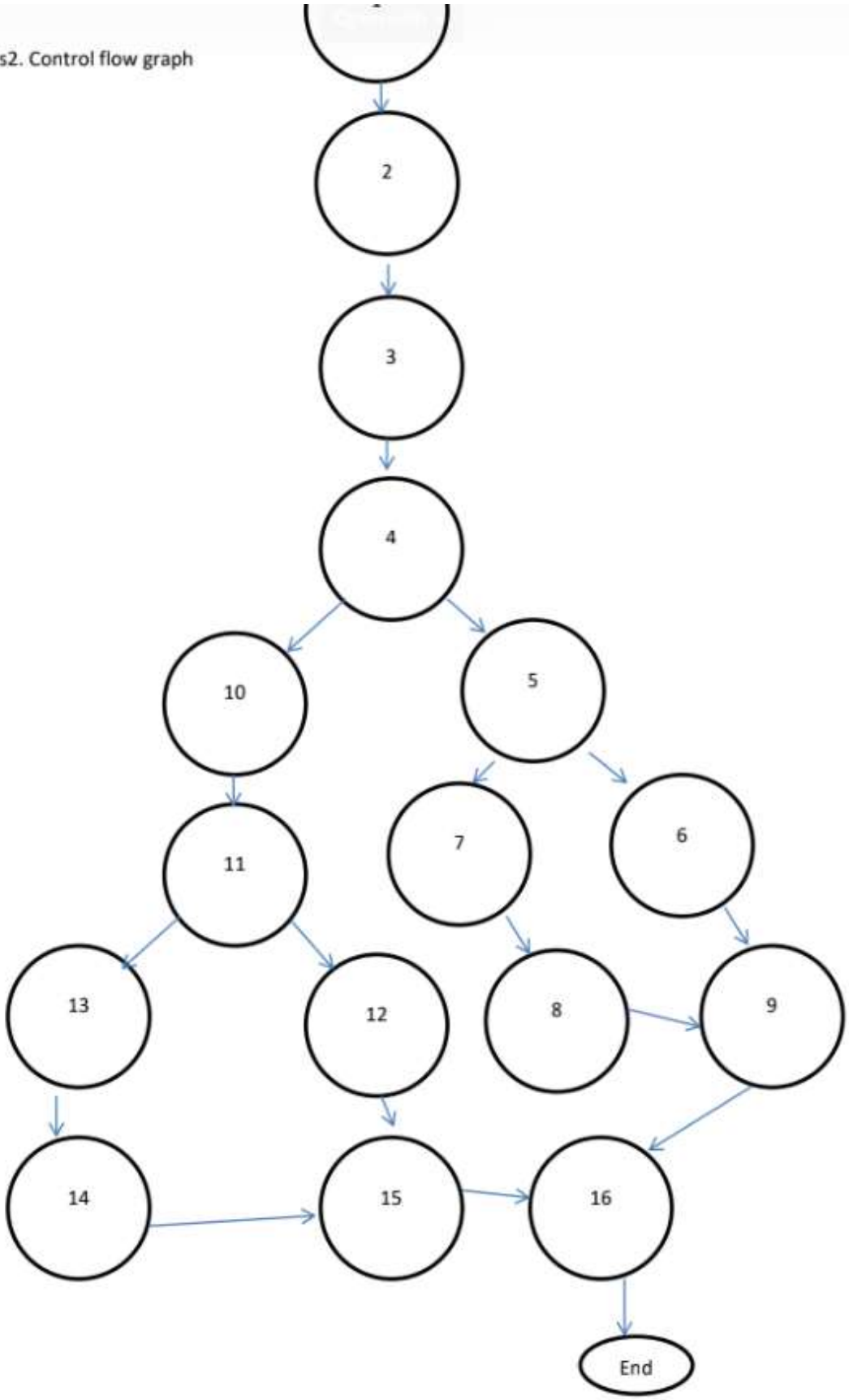
15. endif
16. }

Suppose the program P has been modified (P') as follows.

1. int main(){
2. int a,b,c;
3. input(a,b,c);
4. **if a >= b → Changed Statement**
5. **if a >= c → Changed Statement**
6. print 'a' as highest
7. else
8. print 'c' as highest
9. endif
10. else
11. **if b >=c → Changed Statement**
12. print 'b' as highest
13. else
14. print 'c' as highest
15. endif
16. }

Select the regression test case set T_r for modified program P' using execution trace method. Identify all steps in the process[10M].

Ans2. Control flow graph



Execution Tree

1. T1: Main.stat,main.1,main.2,main.3,main.4,main.5,main.6,main.9,main.16,main.end
2. T2: Main.stat,main.1,main.2,main.3,main.4,main.10,main.main.12,main.15,main.16
3. T3:Main.stat,main.1,main.2,main.3,main.4,main.10,main.11,main.13,main.14,main.15,main.16
4. T4: Main.stat,main.1,main.2,main.3,main.4,main.10,main.11,main.12,main.15,main.16
5. T5:Main.stat,main.1,main.2,main.3,main.4,main.10,main.11,main.13,main.14,main.15,main.16
6. T6: Main.stat,main.1,main.2,main.3,main.4,main.5,main.6,main.9,main.16,main.end

Each testcase carries 1 mark Total=1*6=6

Function	Main()
1	T1,t2,t3,t4,t5,t6
2	T1,t2,t3,t4,t5,t6
3	T1,t2,t3,t4,t5,t6
4	T1,t2,t3,t4,t5,t6
5	T1,t4
6	T1
7	T1
8	T1
9	T1,T6
10	T2,T3,T4,T5
11	T2,T3,T4,T5
12	T2,T4
13	T1,T3,T6
14	T1,T3,T6
15	T1,T2,T3,T4,T6

All correct: 4 marks, Any 5 correct: 2marks, Less than 5 correct:0 marks

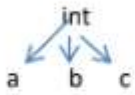
Similar tree for the other changed statements.

To cover	We have to execute
4	T1,T2,T3,T4,T5,T6
5	T1,T4,T6
11	T2,T3,T4,T5

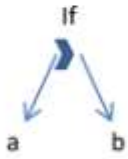
Total= 1*3 marks

Syntax Tree:

- For "INT"



- For "IF"



- For "Print"

Total= 4 marks

Syntax tree for changed program

Total = 4 marks

Similar tree for the other changed statements.

To cover	We have to execute
4	T1,T2,T3,T4,T5,T6
5	T1,T4,T6
11	T2,T3,T4,T5

Total= 1*3 marks