Summer-2020

1(a).Define graph. Write down the set representation of the graph mentioned in Figure-1. 2.0
Answer: Any Graph G(N, N_0, N_f, E), can be defined as-

- A set *N* of nodes, *N* is not empty
- A set N_0 of initial nodes, N_0 is not empty and $N_0 \subseteq N$
- A set N_f of final nodes, N_f is not empty and $N_f \subseteq N$
- A set *E* of edges, each edge from one node to another (n_i, n_j), where *i* is predecessor, *j* is successor and E $\subseteq NxN$

(b). Draw the control flow graph for each of the following code snippet. 3.0

```
x = 0;
while (x < y)
{
  y = f (x, y);
  if (y == 0)
  {
    break;
  } else if (y < 0)
  {
    y = y*2;
    continue;
  }
  x = x + 1;
}
```

(i)

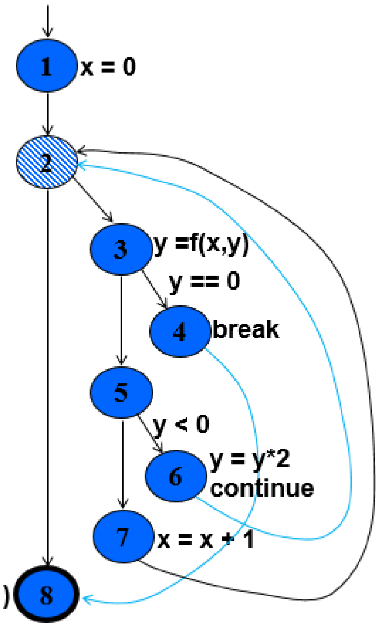
```
/**
  * Return index of node n at the
  * first position it appears,
  * -1 if it is not present
*/
public int indexOf (Node n)
{
 if(n!=null)
 {
   for (int i=0; i < path.size(); i++)
      if (path.get(i).equals(n))
         return i;
 }
   return -1;
}
```

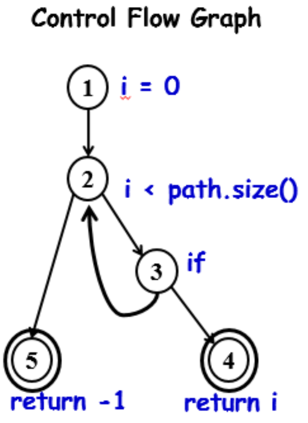
(ii)

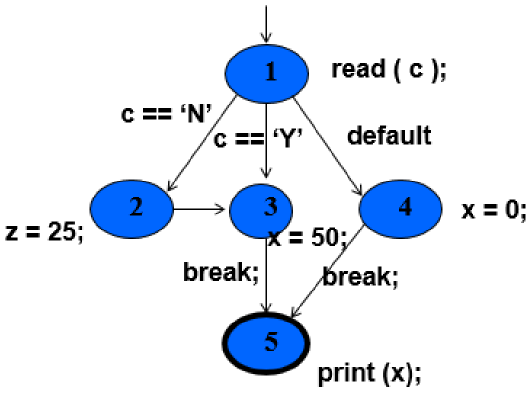
```
read ( c) ;
switch ( c )
{
  case 'N':
    z = 25;
  case 'Z':
    z += 20;
  case 'Y':
    x = 50;
    break;
  default:
    x = 0;
    break;
}
```

(iii)

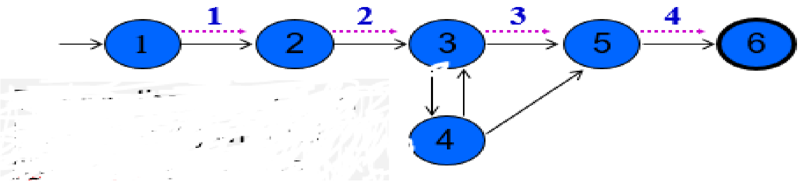
Answer:

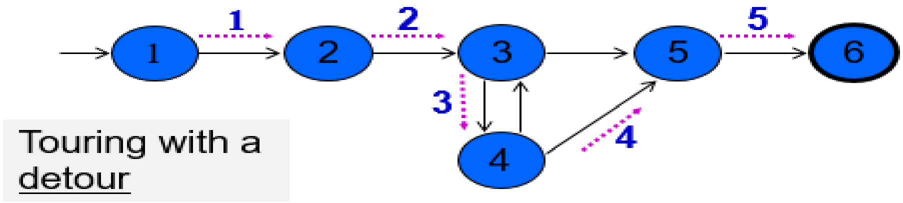






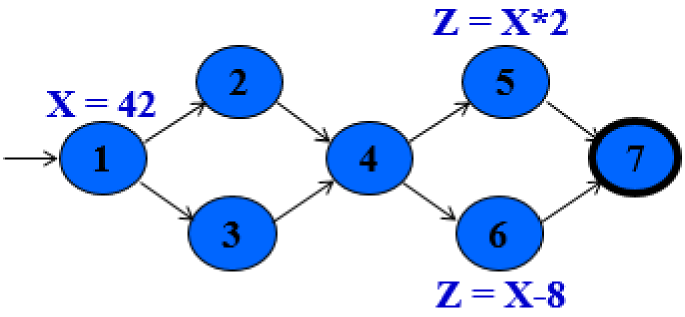
(c). Define detour and def-clear path with appropriate example. 2.0
Answer: **Tour With Detours :** A test path p tours subpath q with detours iff every node in q is also in p in the same order.





<u>def-clear path:</u>A path from l_i to l_j is def-clear with respect to variable v

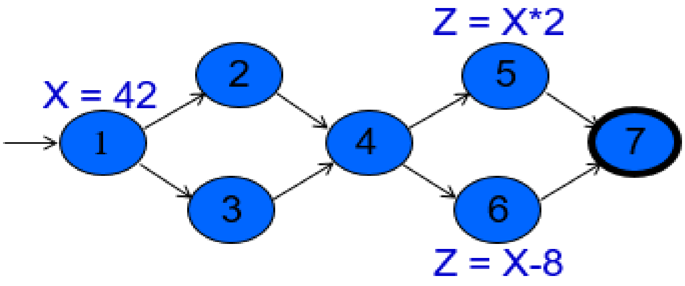
if v is not given another value on any of the nodes or edges in the path



1->2->4->5 & 1->3->4->6

(d). Define def and use. How can we determine the defs of a particular variable from the source code? 3.0

- Definition (def) : A location where a value for a variable is stored into memory
- Use : A location where a variable's value is accessed



Defs: def (1) = {**X** }

def (5) = { **Z** }

def (6) = { **Z** }

Uses: use (5) = {**X** }

use (6) = {**X** }

2. a) Define each of the followings:

i. Error

ii. Failure

iii. Mutation Testing

iv. Happy Path Testing

v. Stress Testing

vi. Regression Testing

Error: An incorrect internal state that is the manifestation of some fault

Failure: A static defect in the software

Mutation Testing: Mutation Testing is a type of white box testing in which the source code of one of the program is changed and verifies whether the existing test cases can identify these defects in the system.

The change in the program source code is very minimal so that it does not impact the entire application, only the specific area having the impact and the related test cases should able to identify those errors in the system.

Happy path testing: The objective of Happy Path Testing is to test an application successfully on a positive flow.

It does not look for negative or error conditions.

The focus is only on the valid and positive inputs through which application generates the expected output.

Stress testing: This testing is done when a system is stressed beyond its specifications in order to check how and when it fails.

This is performed under heavy load like putting large number beyond storage capacity, complex database queries, continuous input to the system or database load.

Regression testing: Testing an application as a whole for the modification in any module or functionality is termed as Regression Testing.

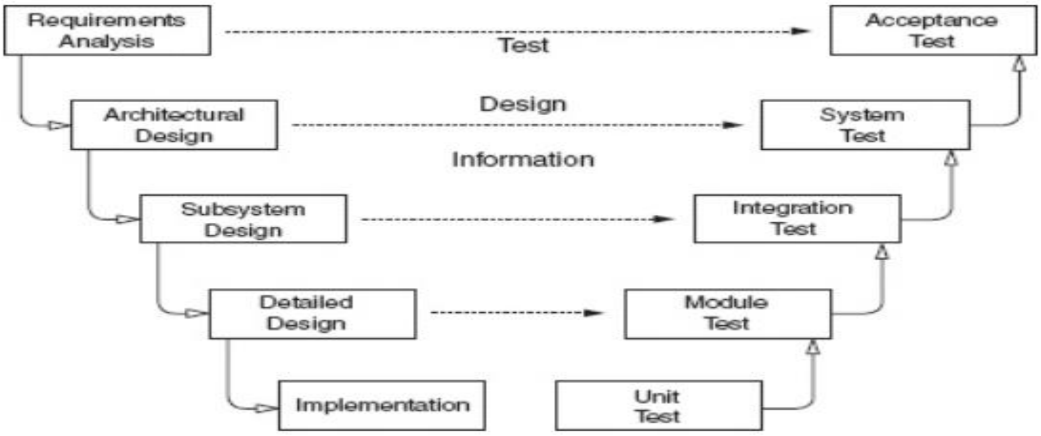
It is difficult to cover all the system in Regression Testing, so typically Automation Testing Tools are used for these types of testing.

(b). Give a comparison between functional testing and non-functional testing. 3.0

- Functional testing verifies each function/feature of the software whereas Non Functional testing verifies non-functional aspects like performance, usability, reliability, etc.
- Functional testing can be done manually whereas Non Functional testing is hard to perform manually.
- Functional testing is based on customer's requirements whereas Non Functional testing is based on customer's expectations.
- Functional testing has a goal to validate software actions whereas Non Functional testing has a goal to validate the performance of the software.
- A Functional Testing example is to check the login functionality whereas a Non Functional testing example is to check the dashboard should load in 2 seconds.
- Functional describes what the product does whereas Non Functional describes how the product works.
- Functional testing is performed before the non-functional testing.

(c). With necessary diagram briefly describe the software testing V-Model. 4.0

Answer:



1.The *requirements analysis* phase of software development captures the customer's needs.

Acceptance testing is designed to determine whether the completed software in fact meets these needs. In other words, acceptance testing probes whether the software does what the users want.

- Acceptance testing must involve users or other individuals who have strong domain knowledge.

2.The *architectural design* phase of software development chooses *components and connectors* that together realize a system whose specification is intended to meet the previously identified requirements.

System testing is designed to determine whether the assembled system meets its specifications. It assumes that the pieces work individually, and asks if the system works as a whole.

- This level of testing usually looks for design and specification problems.
- It is a very expensive place to find lower-level faults and is usually not done by the programmers, but by a separate testing team

3. The *subsystem design* phase of software development specifies the structure and behavior of **subsystems**, each of which is intended to satisfy some function in the overall architecture. Often, the subsystems are adaptations of previously developed software.

Integration testing is designed to assess whether the interfaces between modules (defined below) in a subsystem have consistent assumptions and communicate correctly.

- Integration testing must assume that modules work correctly.
- Integration testing is usually the responsibility of members of the development team

4.The *detailed design* phase of software development determines the structure and behavior of **individual modules**. A *module* is a collection of related units that are assembled in a file, package, or class.

- This corresponds to a file in C, a package in Ada, and a class in C++ and Java.

Module testing is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures.

- Most software development organizations make module testing the responsibility of the programmer; hence the common term *developer testing*.

5.*Implementation* is the phase of software development that actually produces code. A program *unit*, or procedure, is one or more contiguous program statements, with a name that other parts of the software use to call it.

- Units are called functions in C and C++, procedures or functions in Ada, methods in Java, and subroutines in Fortran.

Unit testing is designed to assess the units produced by the implementation phase and is the "lowest" level of testing.

- As with module testing, most software development organizations make unit testing the responsibility of the programmer, again, often called developer testing.

3.(a). Define input domain. Write down the properties of domain partitioning. 1.5

Answer: The input domain for a program contains all the possible inputs to that program.For even small programs, the input domain is so large that it might as well be infinite

Properties of Partitions:

- If the partitions are not complete or disjoint, that means the partitions have not been considered carefully enough
- They should be reviewed carefully, like any design
- Different alternatives should be considered
- We model the input domain in five steps …

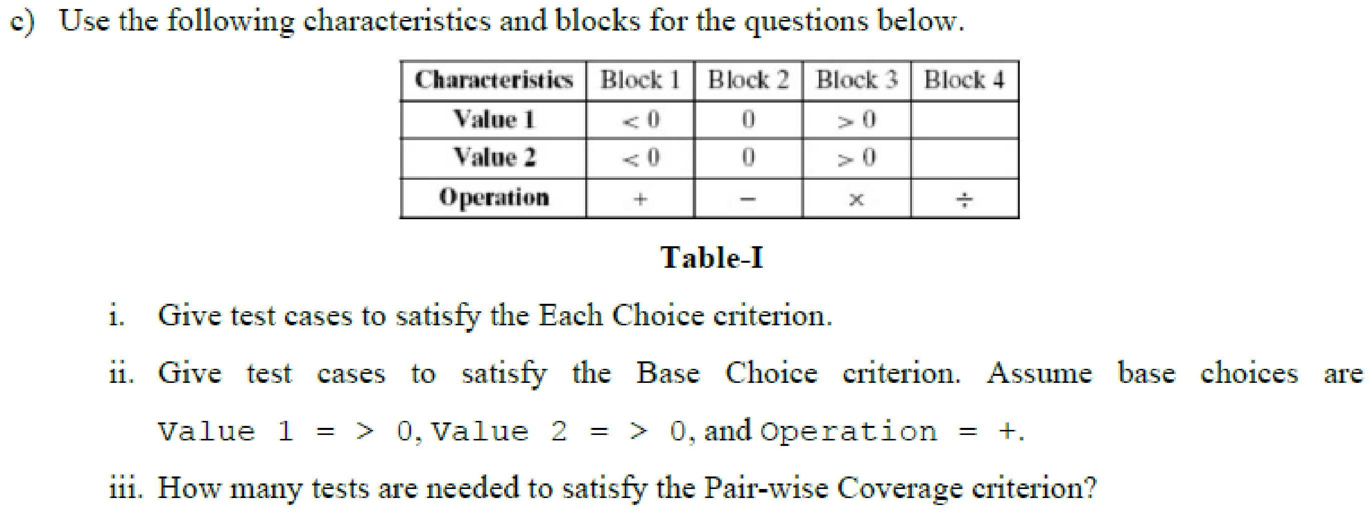
  - Steps 1 and 2 move us from the implementation abstraction level to the design abstraction level (from chapter 2)
  - Steps 3 & 4 are entirely at the design abstraction level
  - Step 5 brings us back down to the implementation abstraction level

(b). Define Coverage Criteria. Write down the advantages of Input Space Partitioning. 1.5

Answer: Coverage criteria. **To measure what percentage of code has been exercised by a test suite**, one or more coverage criteria are used. Coverage criteria are usually defined as rules or requirements, which a test suite needs to satisfy.

advantages of Input Space Partitioning:

- Can be equally applied at several levels of testing
  - Unit
  - Integration
  - System
- Relatively easy to apply with no automation
- Easy to adjust the procedure to get more or fewer tests
- No implementation knowledge is needed
  - Just the input space



c) Use the following characteristics and blocks for the questions below.

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	< 0	0	> 0	
Value 2	< 0	0	> 0	
Operation	+	−	×	÷

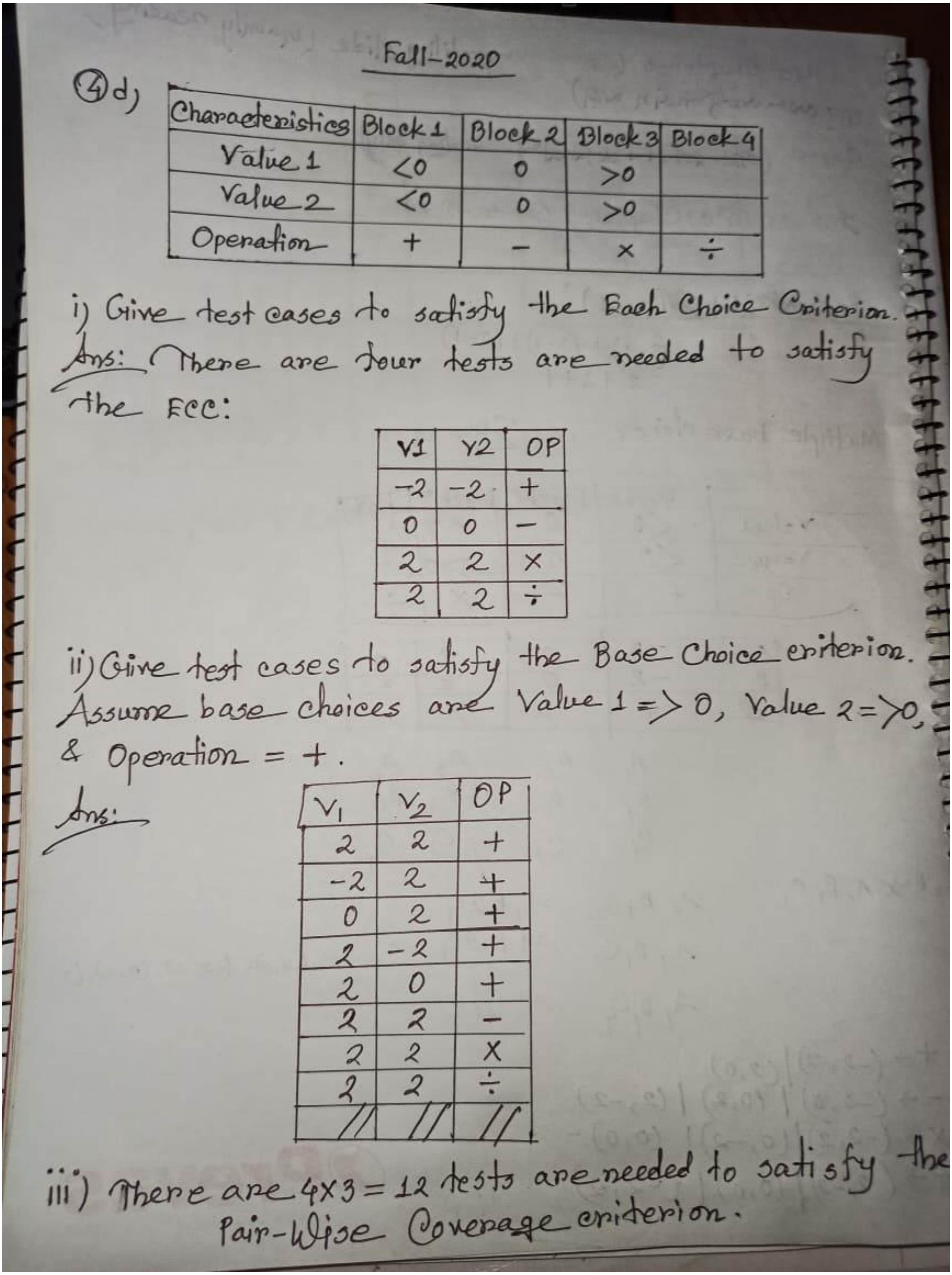
Table-I

i. Give test cases to satisfy the Each Choice criterion.

ii. Give test cases to satisfy the Base Choice criterion. Assume base choices are `Value 1 = > 0`, `Value 2 = > 0`, and `Operation = +`.

iii. How many tests are needed to satisfy the Pair-wise Coverage criterion?

Answer:



Fall-2020

4 d)

Characteristics	Block 1	Block 2	Block 3	Block 4
Value 1	<0	0	>0	
Value 2	<0	0	>0	
Operation	+	−	×	÷

i) Give test cases to satisfy the Each Choice Criterion.

Ans: There are four tests are needed to satisfy the ECC:

V1	V2	OP
−2	−2	+
0	0	−
2	2	×
2	2	÷

ii) Give test cases to satisfy the Base Choice criterion. Assume base choices are Value 1 => 0, Value 2 =>0, & Operation = +.

Ans:

V_1	V_2	OP
2	2	+
−2	2	+
0	2	+
2	−2	+
2	0	+
2	2	−
2	2	×
2	2	÷
//	//	//

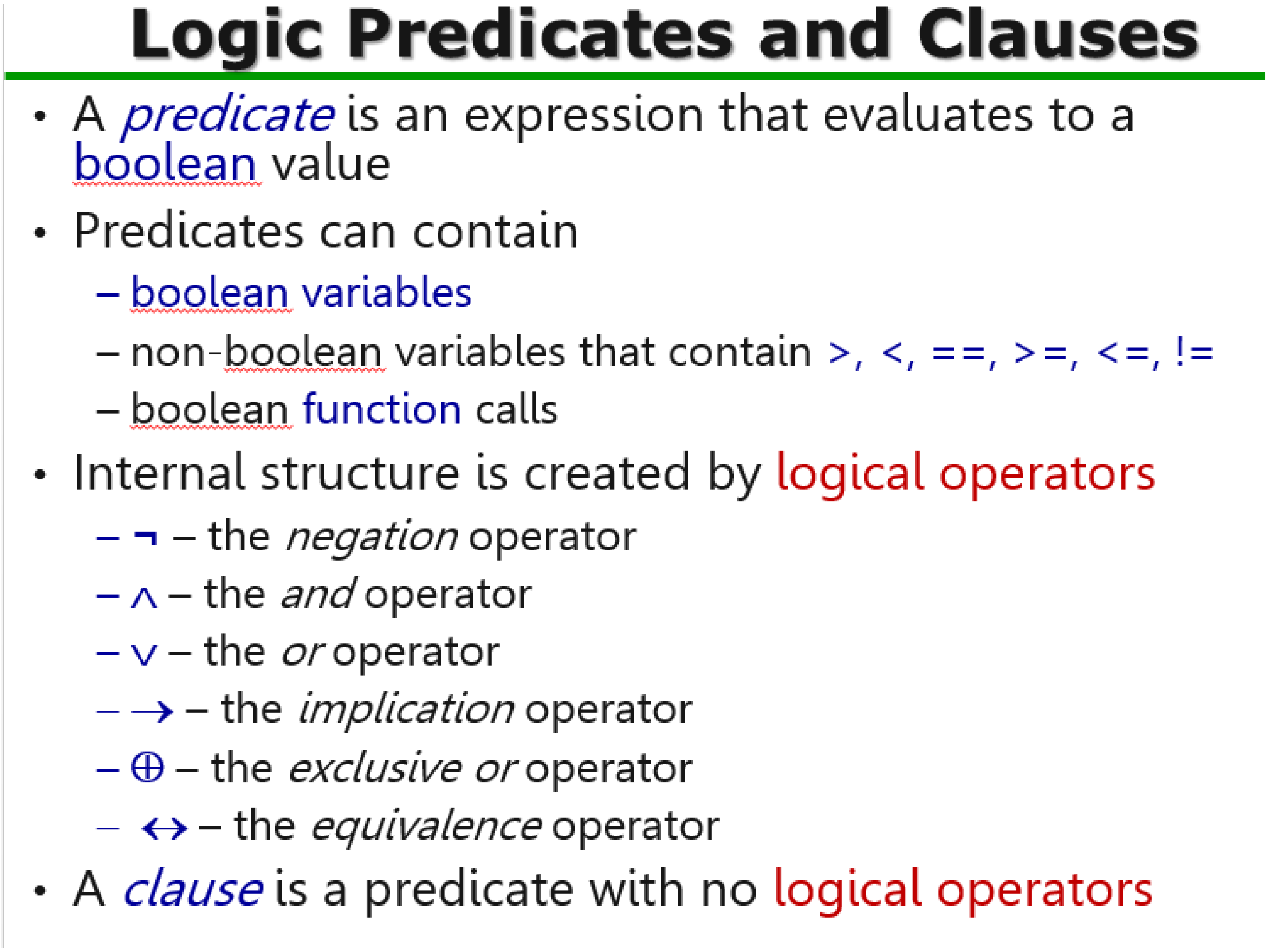
iii) There are 4X3 = 12 tests are needed to satisfy the Pair-Wise Coverage criterion.

(d). Briefly describe the five steps of input domain modeling. 4.0

- Answer: Step 1 : Identify testable functions
  - Individual methods have one testable function
  - Methods in a class often have the same characteristics
  - Programs have more complicated characteristics—modeling documents such as UML can be used to design characteristics
  - Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc.
- Step 2 : Find all the parameters
  - Often fairly straightforward, even mechanical
  - Important to be complete
  - Methods : Parameters and state (non-local) variables used
  - Components : Parameters to methods and state variables
  - System : All inputs, including files and databases
- Step 3 : Model the input domain
  - The domain is scoped by the parameters
  - The structure is defined in terms of characteristics
  - Each characteristic is partitioned into sets of blocks
  - Each block represents a set of values
  - This is the most creative design step in using ISP
- Step 4 : Apply a test criterion to choose combinations of values
  - A test input has a value for each parameter
  - One block for each characteristic
  - Choosing all combinations is usually infeasible
  - Coverage criteria allow subsets to be chosen
- Step 5 : Refine combinations of blocks into test inputs
  - Choose appropriate values from each block

4.(a). Define Logic Coverage. Write down the sources of logic expressions and predicates. 2.5

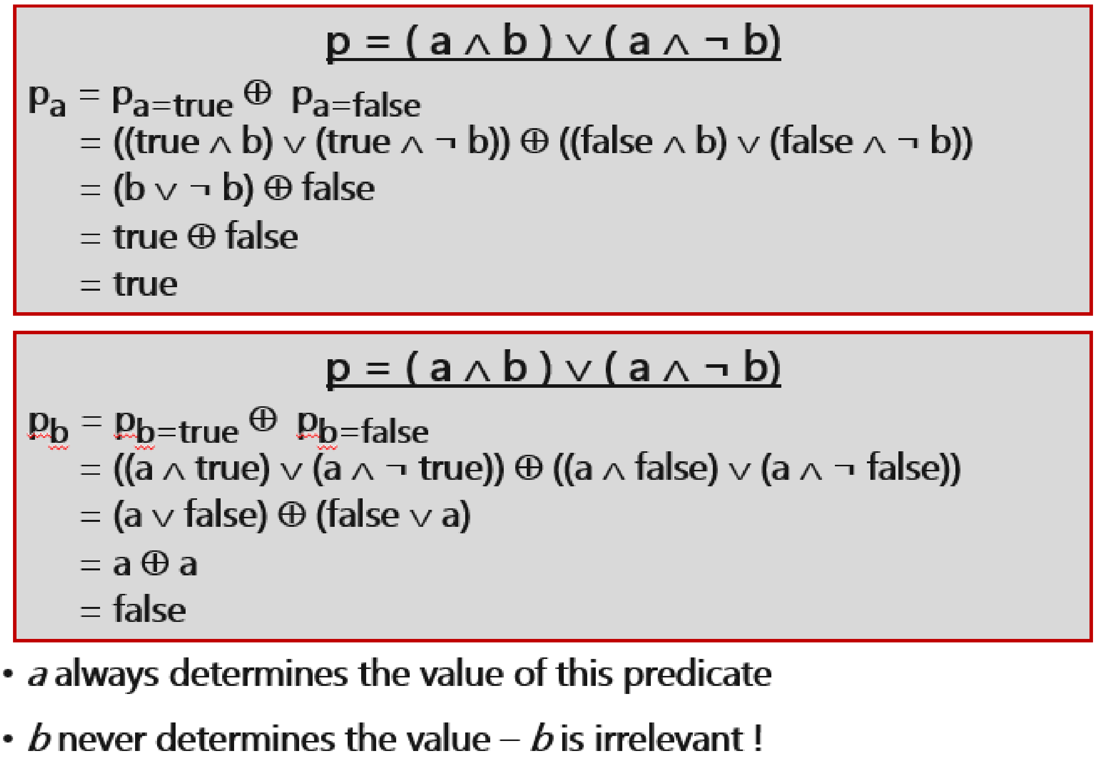
Answer: **Logic corresponds to the internal structure of the code** and this testing is adopted for safety-critical applications such as softwares used in aviation industry. This Test verifies the subset of the total number of truth assignments to the expressions.



Logic Predicates and Clauses

- A *predicate* is an expression that evaluates to a boolean value
- Predicates can contain
  - boolean variables
  - non-boolean variables that contain >, <, ==, >=, <=, !=
  - boolean function calls
- Internal structure is created by logical operators
  - ¬ – the *negation* operator
  - ∧ – the *and* operator
  - ∨ – the *or* operator
  - → – the *implication* operator
  - ⊕ – the *exclusive or* operator
  - ↔ – the *equivalence* operator
- A *clause* is a predicate with no logical operators

b) Define determination in logic coverage. For the logic expression, $p = (x \wedge y) \vee (x \wedge \neg y)$ show that only clause x determines the predicate



$$p = (a \wedge b) \vee (a \wedge \neg b)$$

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} \\ &= ((true \wedge b) \vee (true \wedge \neg b)) \oplus ((false \wedge b) \vee (false \wedge \neg b)) \\ &= (b \vee \neg b) \oplus false \\ &= true \oplus false \\ &= true \end{aligned}$$

$$p = (a \wedge b) \vee (a \wedge \neg b)$$

$$\begin{aligned} p_b &= p_{b=true} \oplus p_{b=false} \\ &= ((a \wedge true) \vee (a \wedge \neg true)) \oplus ((a \wedge false) \vee (a \wedge \neg false)) \\ &= (a \vee false) \oplus (false \vee a) \\ &= a \oplus a \\ &= false \end{aligned}$$

- *a* always determines the value of this predicate
- *b* never determines the value – *b* is irrelevant !

c) Give a comparison between CACC and RACC.

Answer:CACC:

- A more recent interpretation
- Implicitly allows minor clauses to have different values
- Explicitly satisfies (subsumes) predicate coverage

RACC:

- This has been a common interpretation by aviation developers
- RACC often leads to infeasible test requirements
- There is no logical reason for such a restriction

d) Consider the following graph:

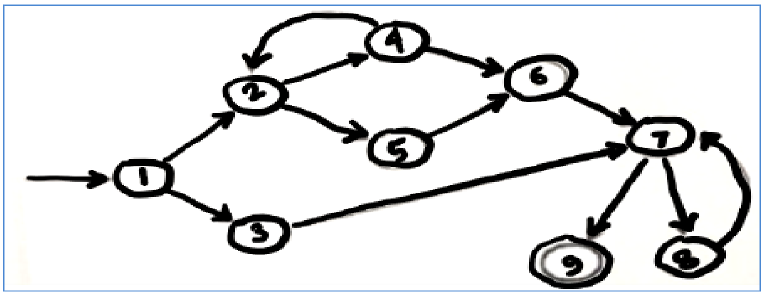


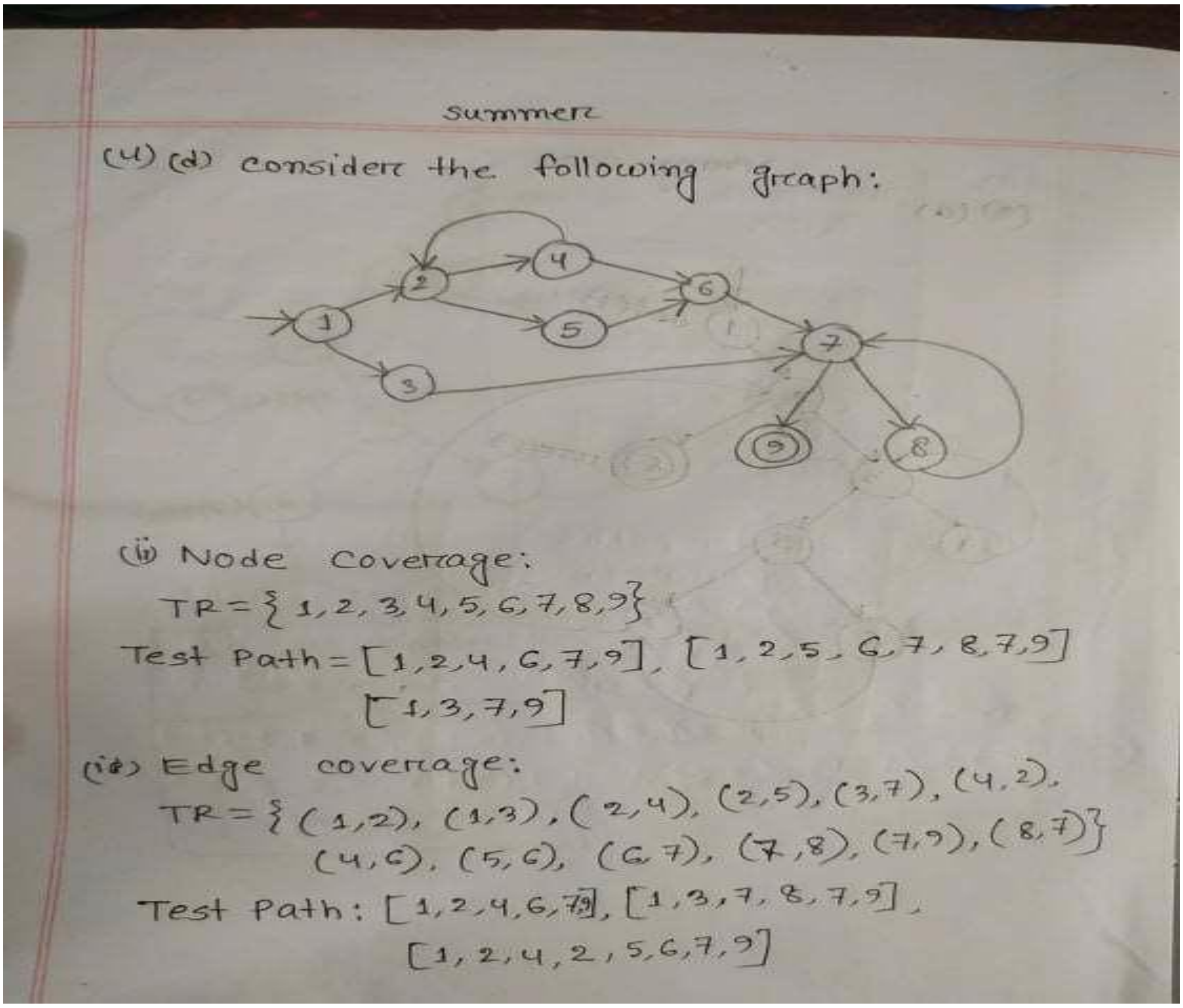
Figure-1

Now write down the test requirements and test paths for each of the following criteria:

i. Edge Coverage

ii. Node Coverage

iii. Edge-pair Coverage



summer

(4) (d) consider the following graph:

(ii) Node coverage:

TR = {1, 2, 3, 4, 5, 6, 7, 8, 9}

Test Path = [1, 2, 4, 6, 7, 9], [1, 2, 5, 6, 7, 8, 7, 9]

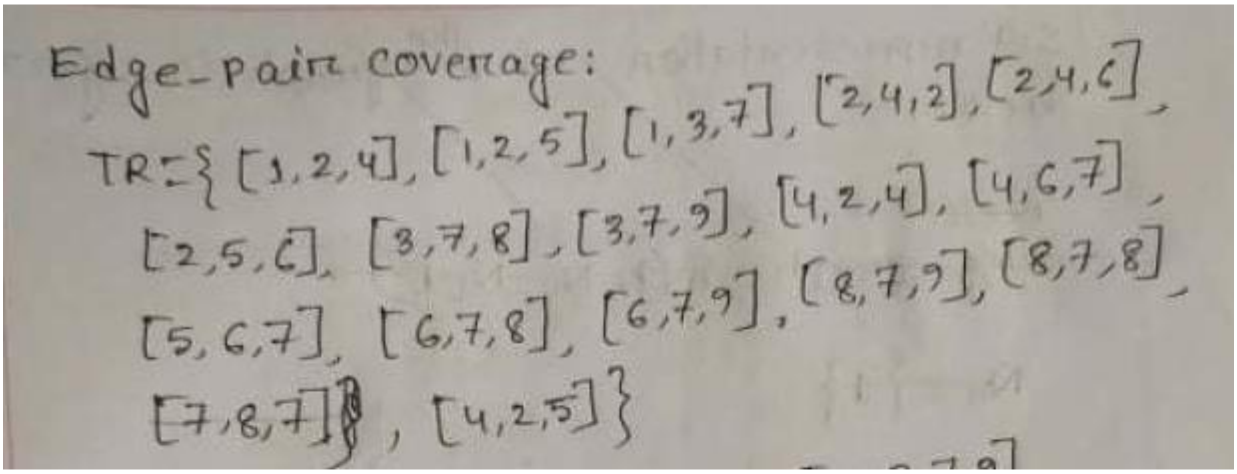
[1, 3, 7, 9]

(i) Edge coverage:

TR = {(1,2), (1,3), (2,4), (2,5), (3,7), (4,2), (4,6), (5,6), (6,7), (7,8), (7,9), (8,7)}

Test Path: [1, 2, 4, 6, 7, 9], [1, 3, 7, 8, 7, 9],

[1, 2, 4, 2, 5, 6, 7, 9]



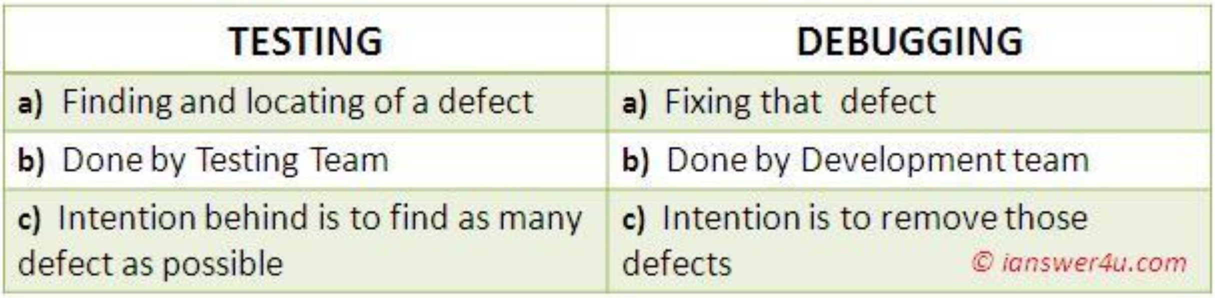
Edge-pair coverage:

TR={[1,2,4], [1,2,5], [1,3,7], [2,4,2], [2,4,6], [2,5,6], [3,7,8], [3,7,9], [4,2,4], [4,6,7], [5,6,7], [6,7,8], [6,7,9], [8,7,9], [8,7,8], [7,8,7], [4,2,5]}

Spring-2020

1. a) Give a comparison between testing and debugging 3

Answer:



TESTING	DEBUGGING
a) Finding and locating of a defect	**a)** Fixing that defect
b) Done by Testing Team	**b)** Done by Development team
c) Intention behind is to find as many defect as possible	**c)** Intention is to remove those defects

© ianswer4u.com

b) Distinguish between Stress Testing and Load Testing. Write down the principles of software testing. 3

Difference between the load testing & stress testing

Load testing	Stress testing
1). Finding the application behaviour with expected load.	1).Finding the application behaviour beyond the expected load.
2). If the build is stable, we can go for load testing	2).If the application is satisfying for load testing objectives; we can go for Stress Testing.
3).Doing this test we can validate the SLA/Performance Testing Objectives	3). By doing this test, we can find the Breaking point of the application.
4).we can find the application behaviour here	4).we can find the short term memory leaks here.

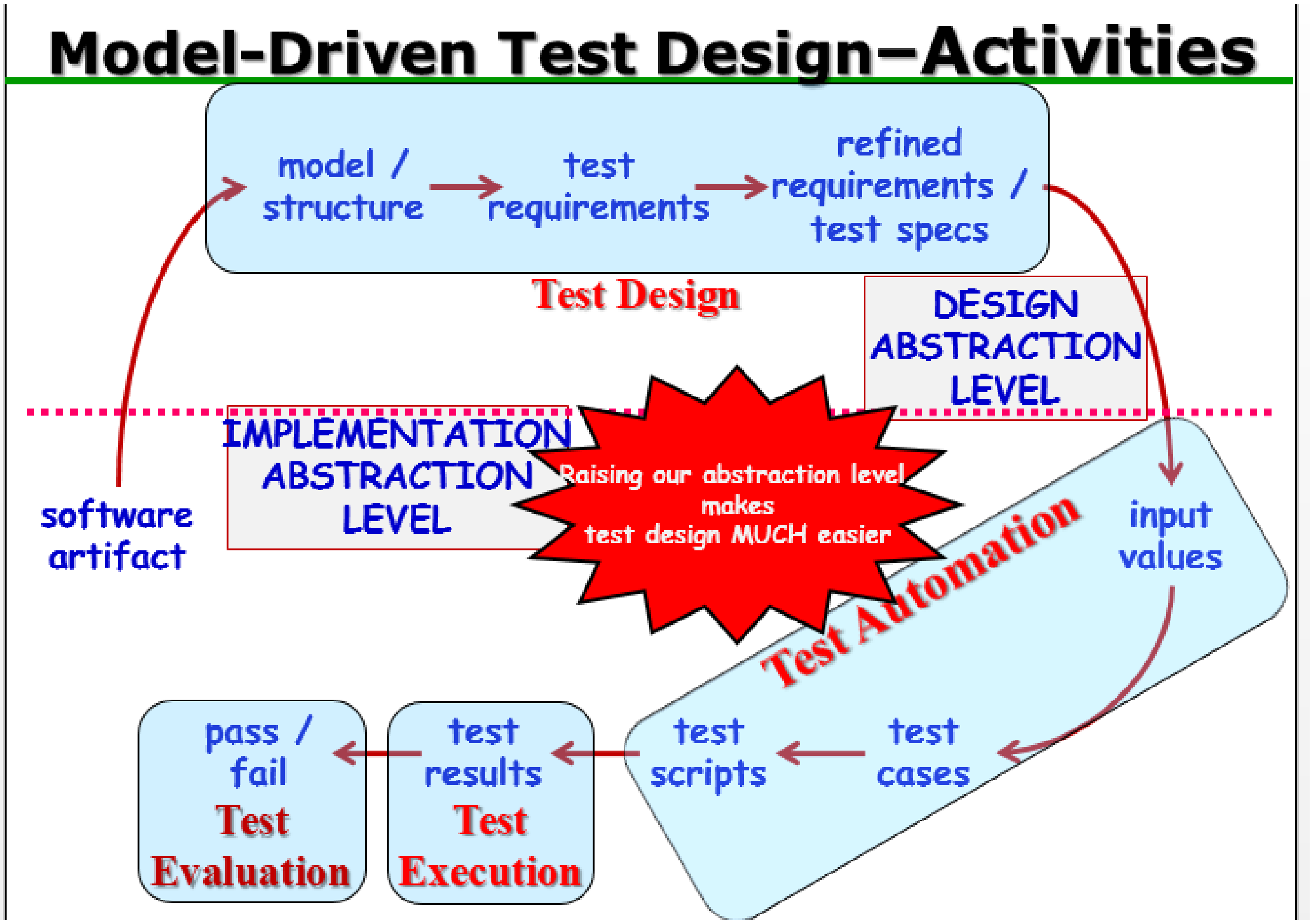
There are seven principles in software testing:

1. Testing shows the presence of defects

2. Exhaustive testing is not possible
3. Early testing
4. Defect clustering
5. Pesticide paradox
6. Testing is context-dependent
7. Absence of errors fallacy

c) Define Coverage Criterion. With necessary diagram briefly describe the MDTD activities. 4

Answer: Coverage criteria give structured, practical ways to search the input space.



Model-Driven Test Design–Activities

1. Test Design — This can be done in either Criteria-Based where Design test values satisfy coverage criteria or other engineering goals or in Human-Based where Design test values based on domain knowledge of the program and human knowledge of testing which is comparatively harder. This the most technical part of the MDTD process better to use experienced developers in this phase.

2. Test Automation — This involves embedding test values to scripts. Test cases are defined based on the test requirements. Test values are chosen such that we can cover a larger part of the application with fewer test cases. We don’t need that much domain knowledge in this phase, however, we need to use technically skilled people.

3. Test Execution — The test engineer will run tests and records the results in this activity. Unlike the previous activities, test execution not required a high skill set such as technical knowledge, logical thinking, or domain knowledge. Since we consider this phase comparatively low risk, we can assign junior

intern engineers to execute the process. But we should focus on monitoring, log collecting activities based on automation tools.

4. Test Evaluation — The process of evaluating the results and reporting to developers. This phase is comparatively harder and we expected to have knowledge in the domain, testing, and User interfaces, and psychology

2. Consider the following information about a graph and answer each of the followings

$N = \{1, 2, 3, 4, 5, 6, 8\}$
$N_o = \{1\}$
$N_f = \{8\}$
$E = \{(1,2), (2,3), (2,8), (3,4), (3,5), (4,3), (5,6), (5,7), (6,7), (7,2)\}$
$\text{def}(1) = \text{def}(4) = use(6) = use(8) = \{x\}$

a) Draw the graph. 2

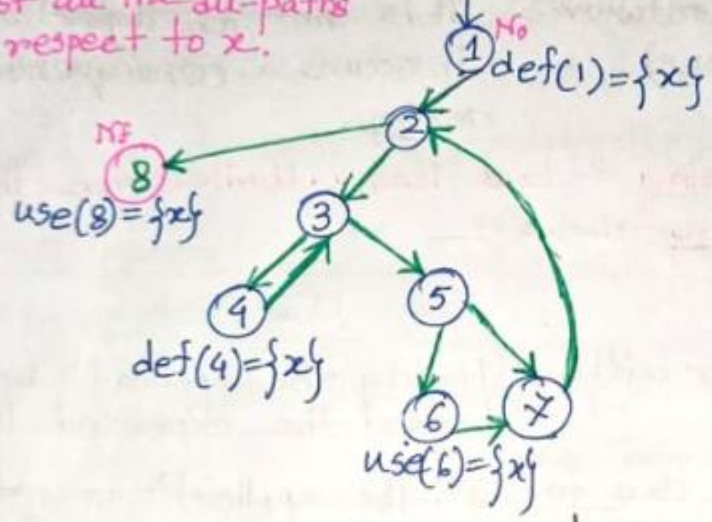
b) List all the du-paths with respect to x. 2

c) List a minimal test set that satisfies all uses coverage with respect to x. 3

d) List a minimal test set that satisfies all-du-paths coverage with respect to x. 3

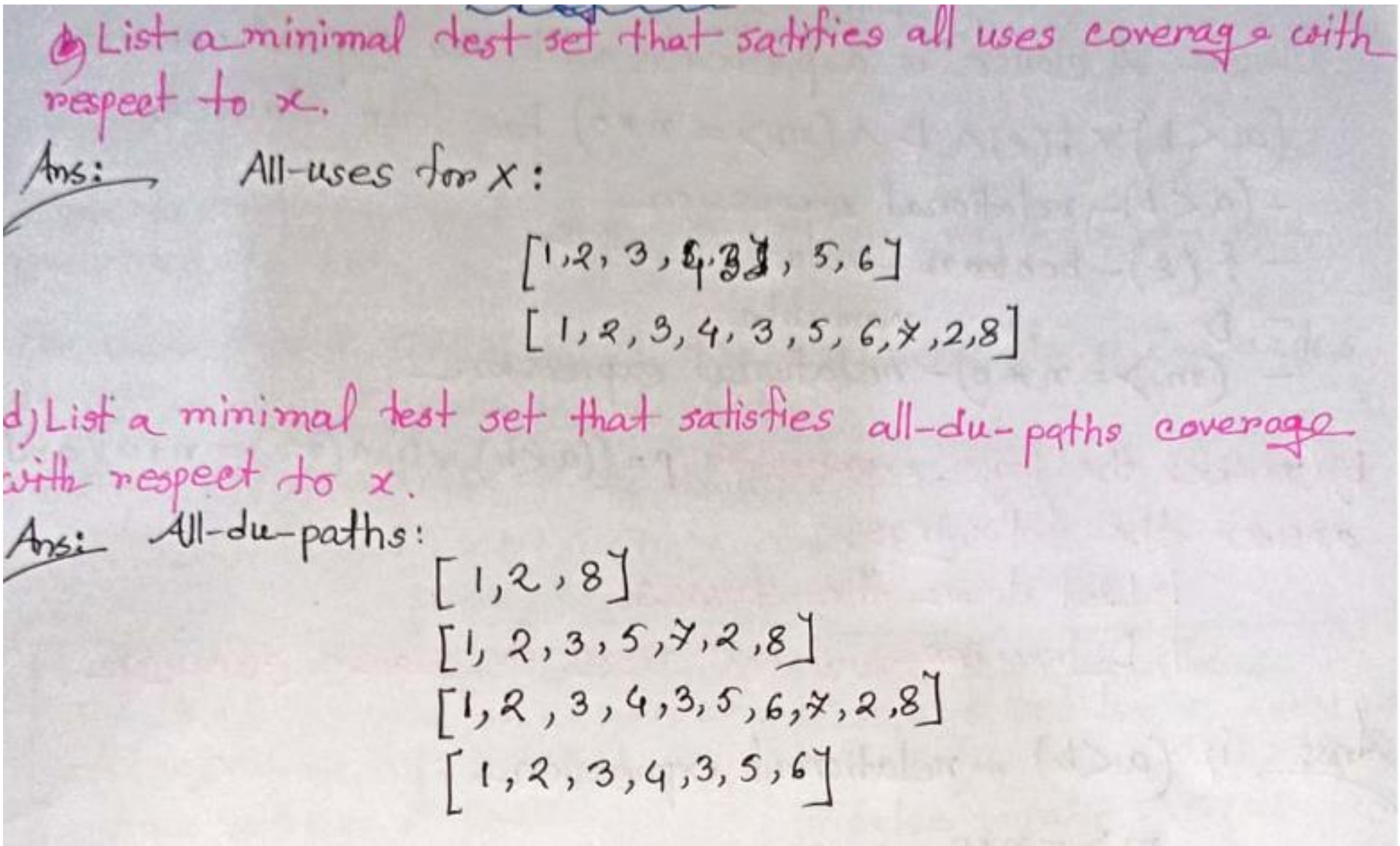
a) Draw the graph.
b) List all the du-paths with respect to x.

Ans:



All du-paths with respect to x :

[1, 2 ,8]
[1, 2, 3 , 5 , 7, 2 , 8]
[1, 2, 3, 4, 3 , 5 , 6 , 7, 2, 8]
[1, 2 , 3 , 5, 6]



c) List a minimal test set that satifies all uses coverage with respect to x.

Ans: All-uses for x :

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

[1,2,3,4,3,5,6,7,2,8]

d) List a minimal test set that satisfies all-du-paths coverage with respect to x.

Ans: All-du-paths :

[1,2,8]

[1,2,3,5,7,2,8]

[1,2,3,4,3,5,6,7,2,8]

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

3. a) Define predicate and clause. 2

b) Consider the logic expression, $p = ((a < b) \vee D) \wedge (m >= n * o)$ and answer the followings: 2

i) List down the clauses

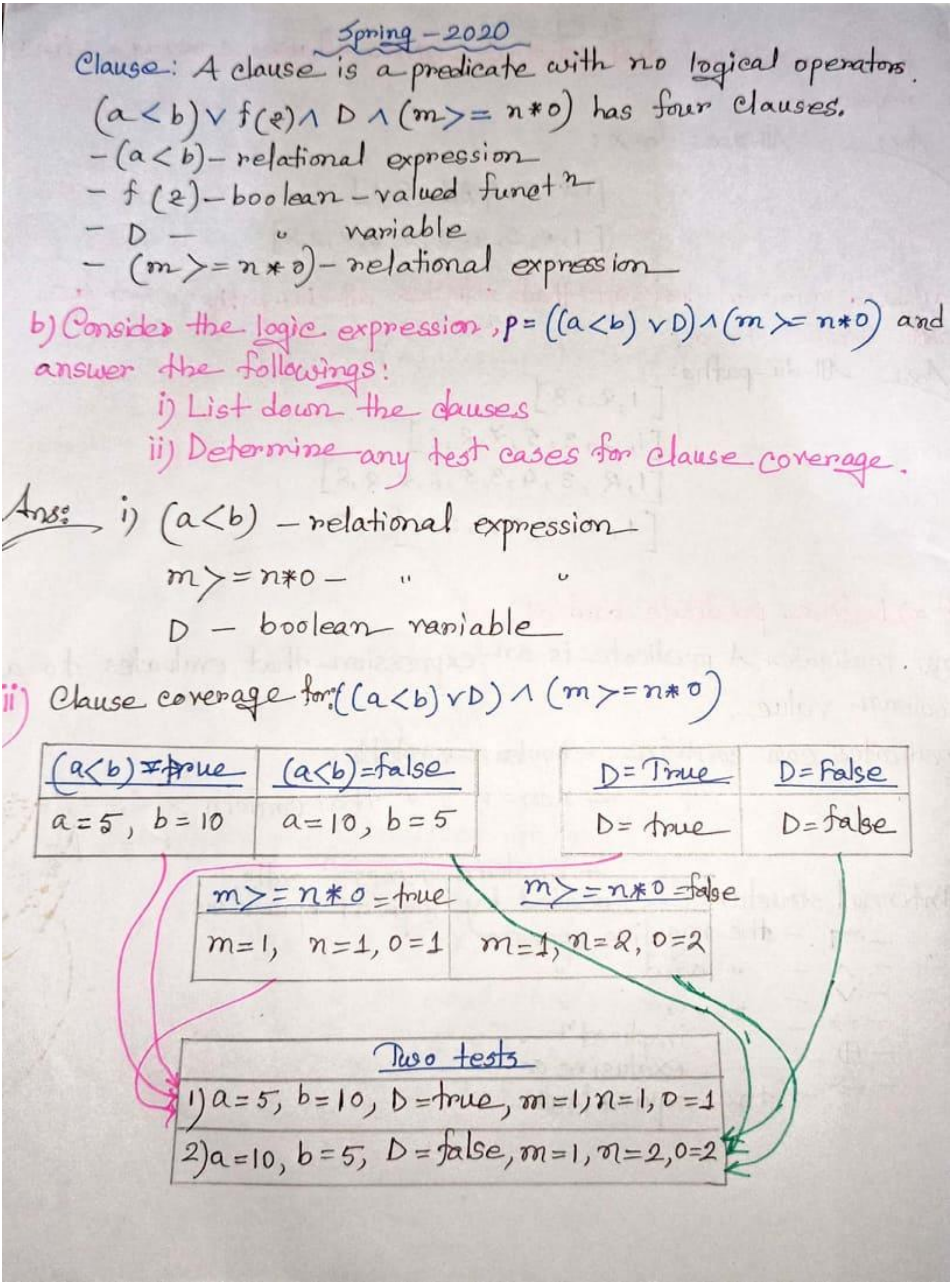
ii) Determine any test cases for clause coverage.

c) Define predicate coverage (PC) and combinatorial coverage (CoC). 2

d) Determine the CACC and RACC pairs of the clauses for the following logic expression: 4

$$p = (\neg a \wedge \neg b) \vee (a \wedge \neg c) \vee (\neg a \wedge c)$$

(a)(b).Answer:



Spring - 2020

Clause: A clause is a predicate with no logical operators.

$(a<b) \vee f(z) \wedge D \wedge (m>=n*o)$ has four clauses.

- $(a<b)$ - relational expression
- $f(z)$ - boolean - valued funct^n
- D - " variable
- $(m>=n*o)$ - relational expression

b) Consider the logic expression, $p=((a<b) \vee D) \wedge (m>=n*o)$ and answer the followings:

i) List down the clauses

ii) Determine any test cases for clause coverage.

Ans: i) $(a<b)$ - relational expression

$m>=n*o$ - " "

D - boolean variable

ii) Clause coverage for: $((a<b) \vee D) \wedge (m>=n*o)$

(a<b)=true	(a<b)=false
a=5, b=10	a=10, b=5

D=True	D=False
D=true	D=false

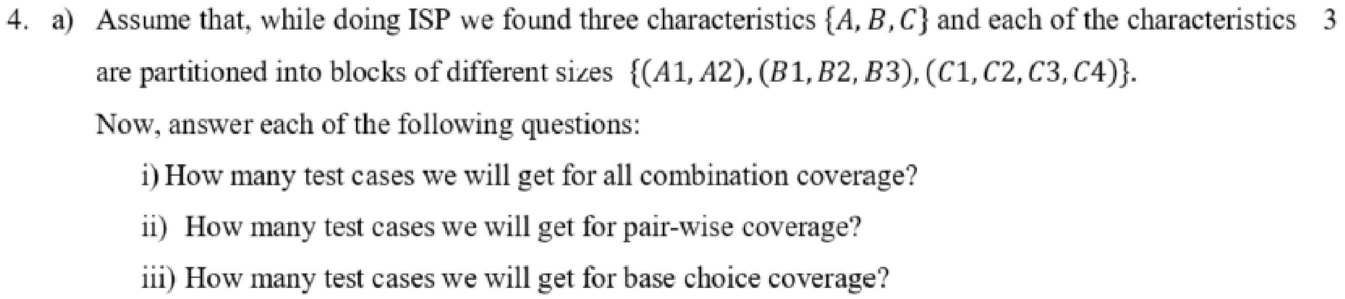
m>=n*o=true	m>=n*o=false
m=1, n=1, o=1	m=1, n=2, o=2

Two tests
1) a=5, b=10, D=true, m=1, n=1, o=1
2) a=10, b=5, D=false, m=1, n=2, o=2

(c). Answer: **Predicate Coverage (PC)** **: For each *p* in *P*, *TR* contains two requirements: *p* evaluates to true, and *p* evaluates to false.**

Combinatorial Coverage (CoC) : For each p in P, TR has test requirements for the clauses in Cp to evaluate to each possible combination of truth values.

(d).



4. a) Assume that, while doing ISP we found three characteristics $\{A, B, C\}$ and each of the characteristics are partitioned into blocks of different sizes $\{(A1, A2), (B1, B2, B3), (C1, C2, C3, C4)\}$. 3

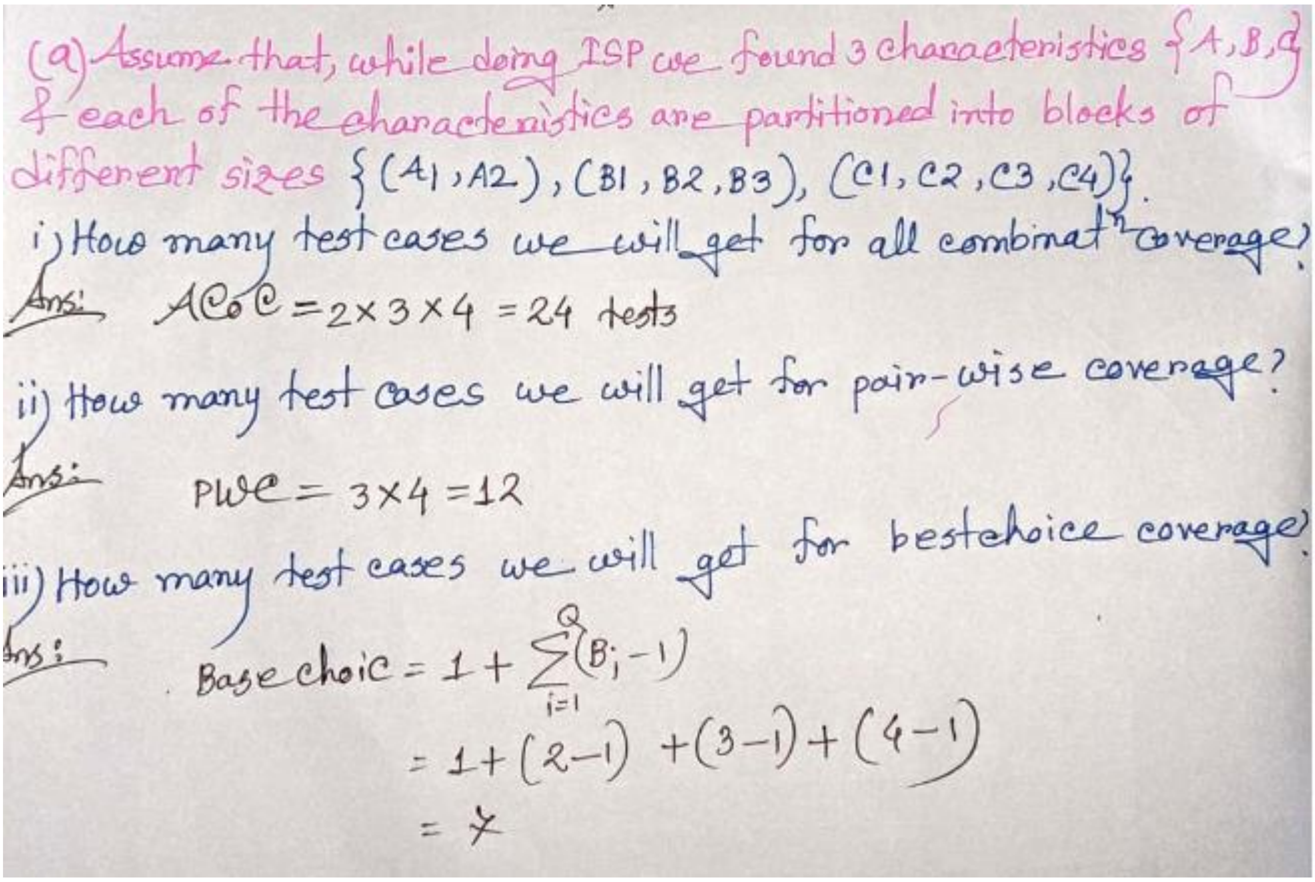
Now, answer each of the following questions:

i) How many test cases we will get for all combination coverage?

ii) How many test cases we will get for pair-wise coverage?

iii) How many test cases we will get for base choice coverage?

Answer:



(a) Assume that, while doing ISP we found 3 characteristics $\{A, B, C\}$ & each of the characteristics are partitioned into blocks of different sizes $\{(A1, A2), (B1, B2, B3), (C1, C2, C3, C4)\}$.

i) How many test cases we will get for all combinat^n coverage?

Ans: ACoC $= 2 \times 3 \times 4 = 24$ tests

ii) How many test cases we will get for pair-wise coverage?

Ans: PWC $= 3 \times 4 = 12$

iii) How many test cases we will get for bestchoice coverage?

Ans: Base choic $= 1 + \sum_{i=1}^{Q} (B_i - 1)$

$= 1 + (2-1) + (3-1) + (4-1)$

$= 7$

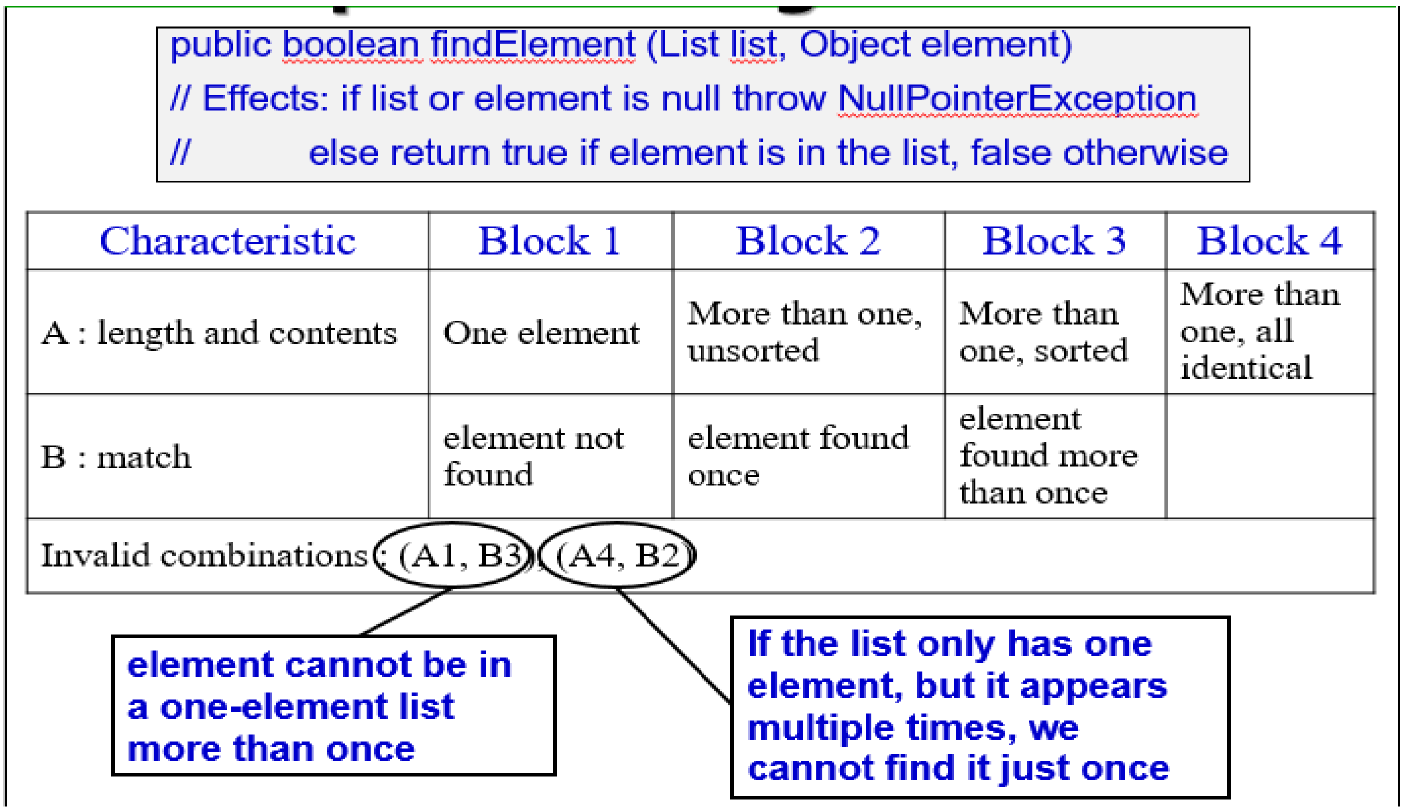
b) Define input domain. Consider the following code segment: 3

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//  else return true if element is in the list, false otherwise
```

Now, give an example of partitioning scheme that will satisfy the following characteristic constraints for the above code snippet and highlight the criteria:

i) A block from one characteristic cannot be combined with a specific block from another.

ii) A block from one characteristic can ONLY BE combined with a specific block form another characteristic.



```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	One element	More than one, unsorted	More than one, sorted	More than one, all identical
B : match	element not found	element found once	element found more than once	
Invalid combinations : (A1, B3) (A4, B2)				

element cannot be in a one-element list more than once

If the list only has one element, but it appears multiple times, we cannot find it just once

c) Define each of the followings with an appropriate example: 4

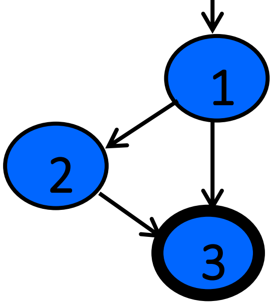
i) Node Coverage

ii) Prime Path

iii) Test Path

iv) T-wise Coverage

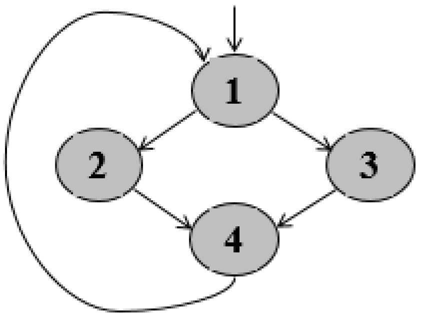
<u>Node Coverage (NC)</u> : Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *path(T)* such that *p* visits *n*.



<u>Node Coverage</u> :? TR = { 1, 2, 3 }

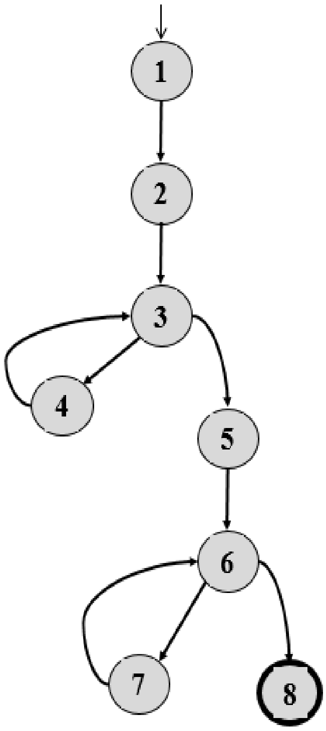
Test Path = [1, 2, 3]

Prime Path : *A simple path that does not appear as a proper subpath of any other simple path*

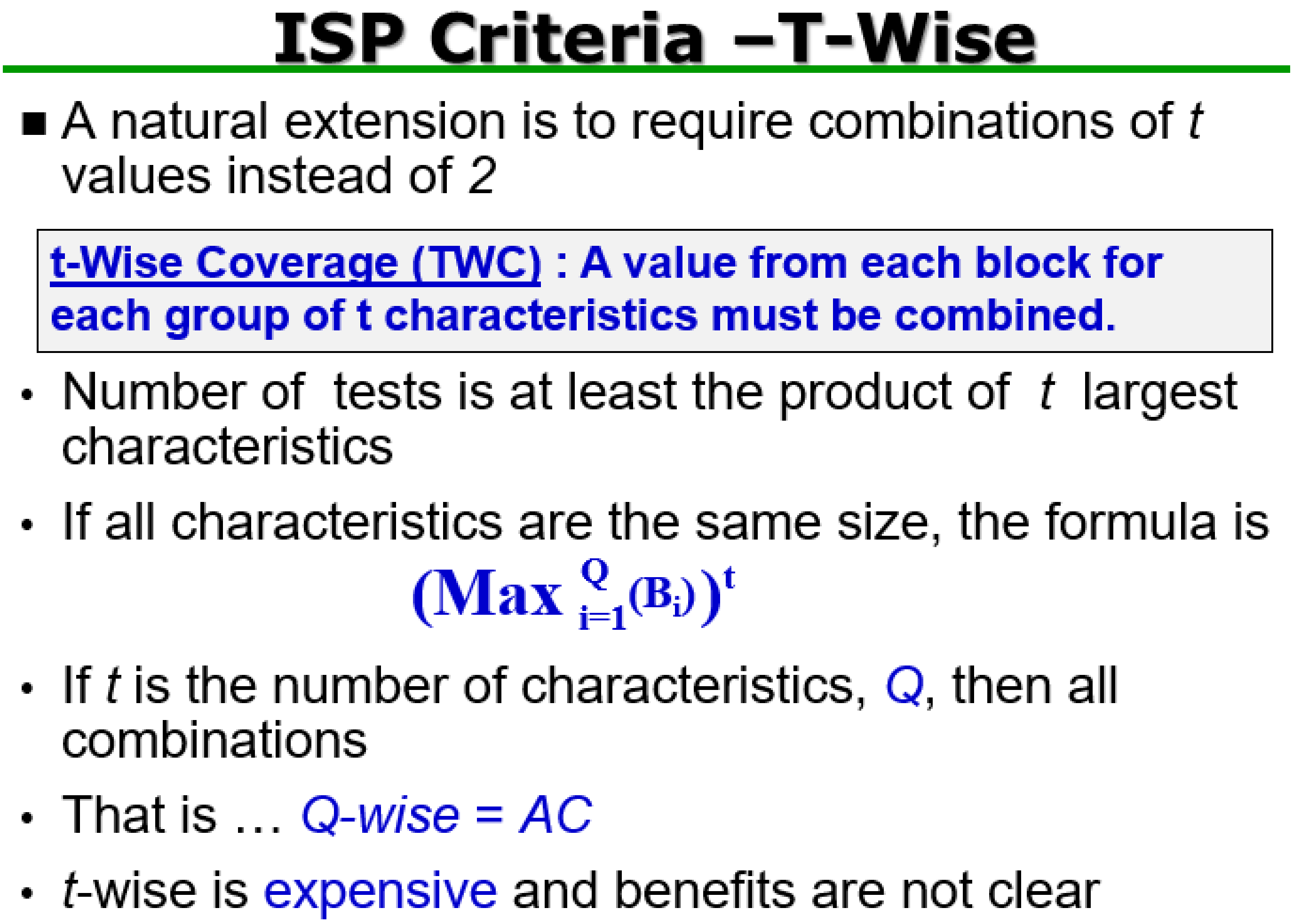


Prime Paths : [2,4,1,2], [2,4,1,3], [1,3,4,1], [1,2,4,1], [3,4,1,2], [4,1,3,4], [4,1,2,4], [3,4,1,3]

Test Path : A path that starts at an initial node and ends at a final node



Edge Coverage	
TR	Test Path
A. [1, 2] B. [2, 3] C. [3, 4] D. [3, 5] E. [4, 3] F. [5, 6] G. [6, 7] H. [6, 8] I. [7, 6]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]



ISP Criteria –T-Wise

- A natural extension is to require combinations of *t* values instead of *2*

t-Wise Coverage (TWC) : A value from each block for each group of t characteristics must be combined.

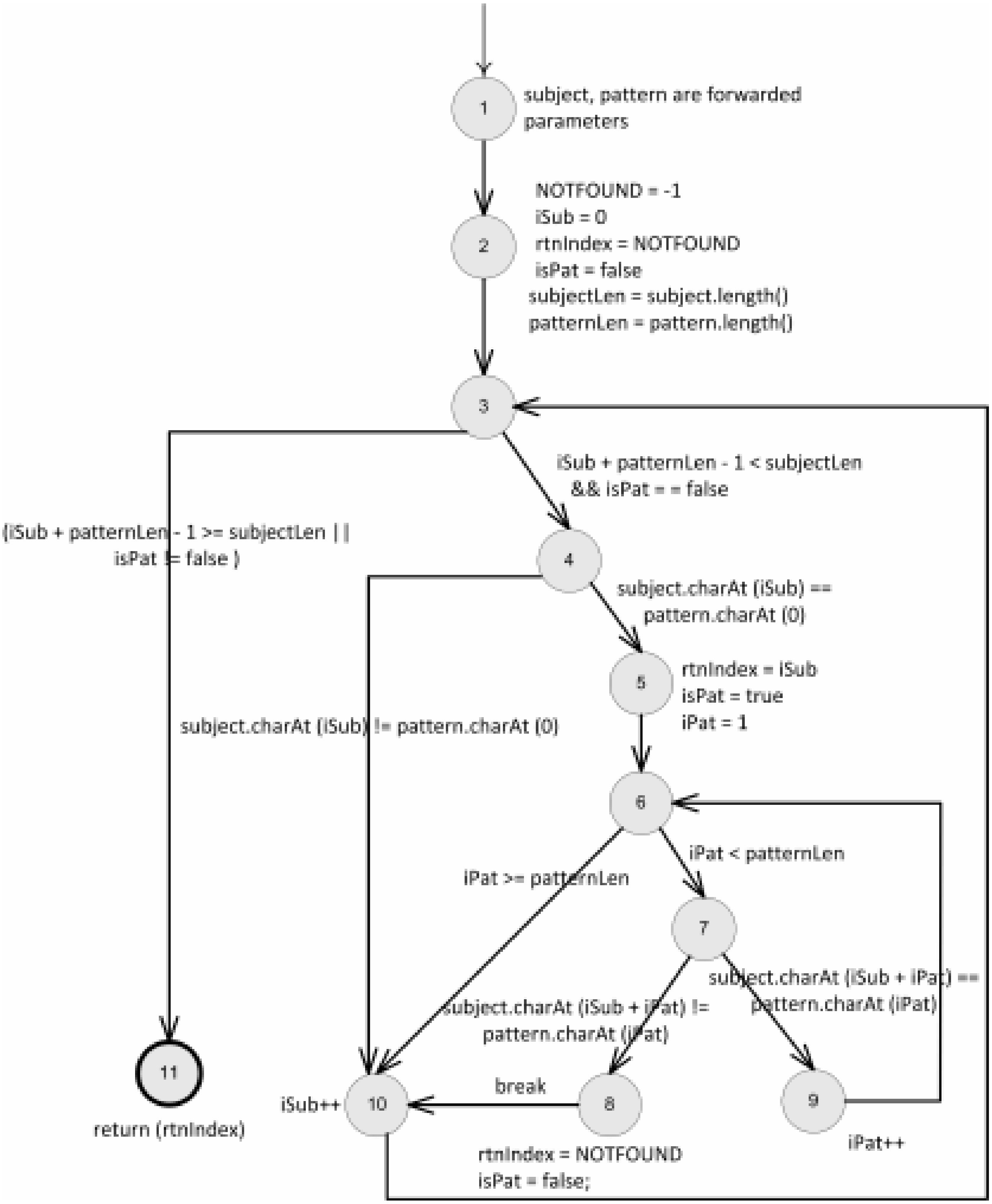
- Number of tests is at least the product of *t* largest characteristics
- If all characteristics are the same size, the formula is

$$(\text{Max}_{i=1}^{Q}(B_i))^t$$

- If *t* is the number of characteristics, Q, then all combinations
- That is … *Q-wise = AC*
- *t*-wise is expensive and benefits are not clear

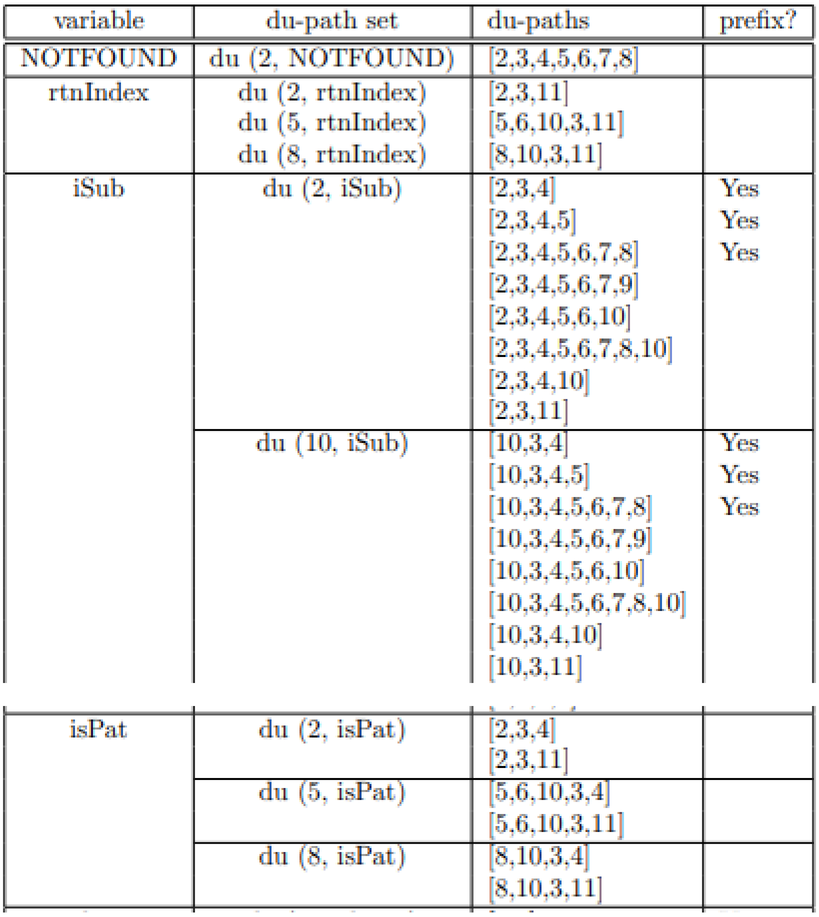
Fall-2020

2.(b).



1
subject, pattern are forwarded parameters
2
NOTFOUND = -1
iSub = 0
rtnIndex = NOTFOUND
isPat = false
subjectLen = subject.length()
patternLen = pattern.length()
3
iSub + patternLen - 1 < subjectLen && isPat = = false
(iSub + patternLen - 1 >= subjectLen | | isPat != false)
4
subject.charAt (iSub) == pattern.charAt (0)
subject.charAt (iSub) != pattern.charAt (0)
5
rtnIndex = iSub
isPat = true
iPat = 1
6
iPat < patternLen
iPat >= patternLen
7
subject.charAt (iSub + iPat) == pattern.charAt (iPat)
subject.charAt (iSub + iPat) != pattern.charAt (iPat)
8
break
rtnIndex = NOTFOUND
isPat = false;
9
iPat++
10
iSub++
11
return (rtnIndex)

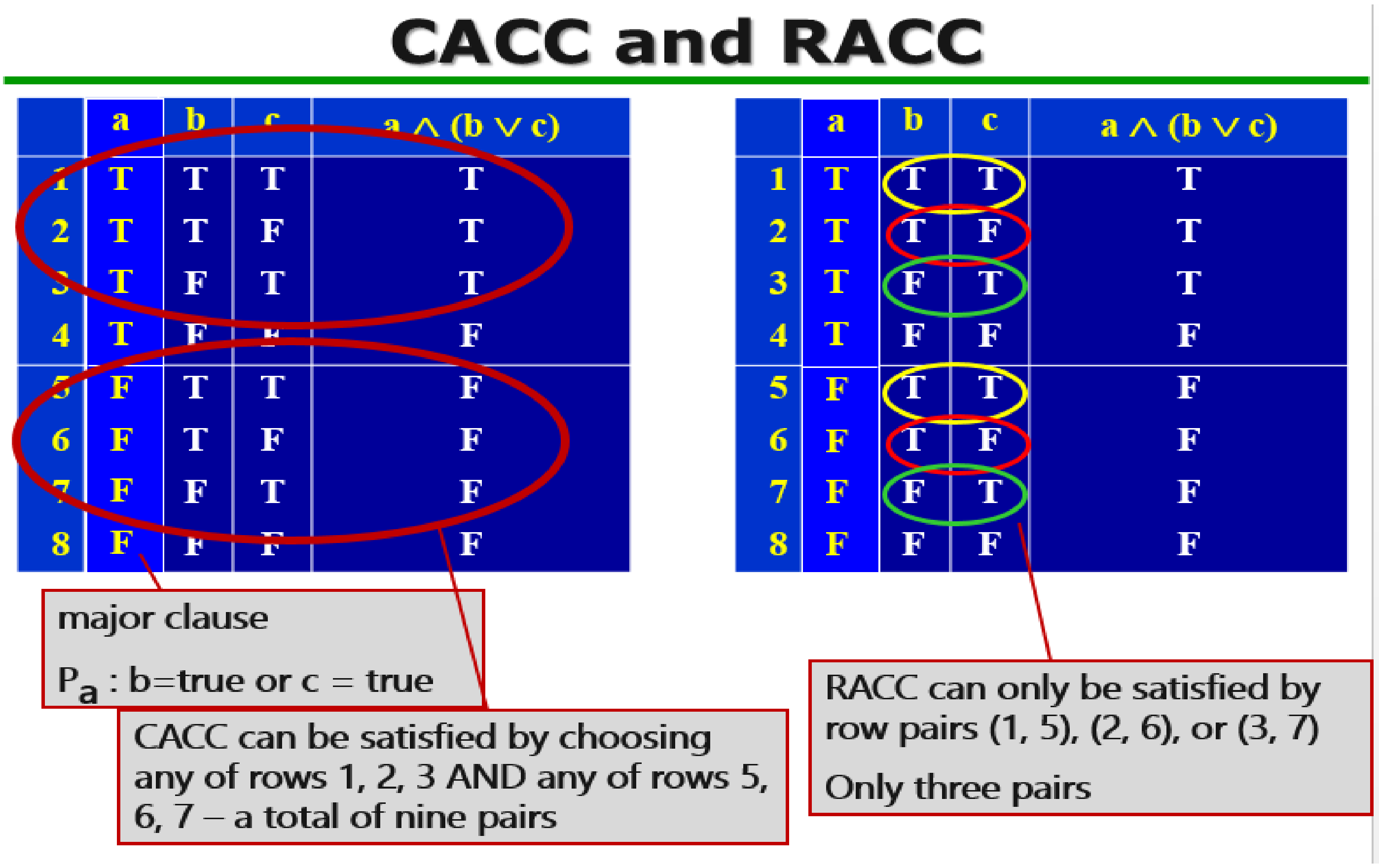
2.(d)



variable	du-path set	du-paths	prefix?
NOTFOUND	du (2, NOTFOUND)	[2,3,4,5,6,7,8]	
rtnIndex	du (2, rtnIndex) du (5, rtnIndex) du (8, rtnIndex)	[2,3,11] [5,6,10,3,11] [8,10,3,11]	
iSub	du (2, iSub)	[2,3,4] [2,3,4,5] [2,3,4,5,6,7,8] [2,3,4,5,6,7,9] [2,3,4,5,6,10] [2,3,4,5,6,7,8,10] [2,3,4,10] [2,3,11]	Yes Yes Yes
	du (10, iSub)	[10,3,4] [10,3,4,5] [10,3,4,5,6,7,8] [10,3,4,5,6,7,9] [10,3,4,5,6,10] [10,3,4,5,6,7,8,10] [10,3,4,10] [10,3,11]	Yes Yes Yes
isPat	du (2, isPat)	[2,3,4] [2,3,11]	
	du (5, isPat)	[5,6,10,3,4] [5,6,10,3,11]	
	du (8, isPat)	[8,10,3,4] [8,10,3,11]	

3.

c) For the expression, $\boldsymbol{E} = \boldsymbol{a} \wedge (\boldsymbol{b} \vee \boldsymbol{c})$ determine the CACC and RACC 3



CACC and RACC

	a	b	c	a ∧ (b ∨ c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

major clause

P_a : b=true or c = true

CACC can be satisfied by choosing any of rows 1, 2, 3 AND any of rows 5, 6, 7 – a total of nine pairs

	a	b	c	a ∧ (b ∨ c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

RACC can only be satisfied by row pairs (1, 5), (2, 6), or (3, 7)

Only three pairs