White-box testing

Agenda

- Introduction
- Code Basics
  - Variables
  - Types
  - Conditionals
  - Loops
  - Functions
- Flowcharts
- White-box testing
  - Cyclomatic Complexity
  - Test Design Techniques
  - Practice
- Data Flow testing
- Conclusion
- References
- Questions

Introduction

This training will be useful for those who:

- Haven't passed ISTQB (foundation level)
- Want to have more arguments in discussions with developers
- Want to know how to write code
- Want to go deep in white-box testing

Code Basics

Variables

- **Variable** is a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information, a value.

```
int a;
string b, c;
a = 234;
b = “Andy”;
c = a + b;
```

Types

- **Basic types**: integer, float, string, char, boolean.

```
int v = 30;
float v = 12.56;
string v = “I love NY”;
char v = ‘H’;
boolean = yes/no, true/false, 1/0
```

Conditionals

- **Conditionals** are features of a programming language which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

```
if (condition) then

    (action)

else

    (action)
```

Conditionals. Types

```
if (condition) then
   (consequent)
else
   (alternative)
end if
```

```
(condition) ? actionOnTrue : actionOnFalse
```

```
if (condition) then
--statements
elseif (condition) then
--more statements
elseif (condition) then
--more statements;
    ...
else
--other statements;
end if;
```

```
switch (someChar)
{
  case 'a': actionA; break;
  case 'x': actionX; break;
  case 'y':------
  case 'z': actionZ; break;
  default: actionNoMatch;
}
```

Conditionals. Examples

```
a = 400;
b = 30;
if (b < a) then
   result = “H
el
   sult = 0;
end if
```

result = “Happy”

```
a = 45; b = –20;
(a > 0  &&  b > 0) || (c) ? resu                    result = “Negative”
```

result=“Negative”

```
age = 27;
  switch (age) {
     case 18:                ng”; break;
     c          sult = “Ok”; break;
     case 60: result = “Old”; break;
     default: result = “Unknown”;
  }
```

result = “Unknown”

```
name1 = Peter;
age1 = 40;
name2 = John;
age2
if (age1 < 40) t
   result = “Choose first
 elseif (age2 < 28 or name2 == “Jo          en
   result = “Choose second”
 end if;
```

result = “Choose second”

Loops

- **Loop** is a sequence of instructions that is continually repeated until a certain condition is reached.

```
while (condition)
{
    statements;
}
```

Loops. Types

```
while (condition)
    {
        do_work();
    }
```

```
do
 {
   do_work();
 }
while (condition);
```

```
for (initialization; condition; increment/decrement)
 {
   do_work();
 }
```

```
for each item in collection:
do something to item
```

Loops. Examples

```
int counter = 3;
int fa          1;

while (c
 {
   factorial *= counter--;
 }
```

factorial = 6 two times

```
int counter = 4;
int factorial = 1;
do
 {
   fac          ounter;
 }
while (counter > 1);
```

factorial = 6 three times

```
int sum = 30;
for (int i = 1; i < 4;
 {
    sum -
 }
```

sum = 36

```
int myint[] = {1, 2, 3, 4, 5};
fo              int)
    {
       i++;
    }
```

myint= {2, 3, 4, 5, 6}

```
while (true) {
                          ;
   }
```

infinite loop

Functions

- **Function** is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed. Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs.
- In different programming languages a function may be called a **procedure**, a **subroutine** , a **routine**, a **method**, or a **subprogram**.

Functions. Example

```
void function1(void)
{
   printf("Hello");
}

int function2(void)
{
   return 5;
}

char function3(int number)
{
   char selection[] = {'S','M','T','W','T','F','S'};
   return selection[number];
}

void main
{
   function1();
   a = 30 + function2();
   c = function3(3);
}
```

Flowcharts

Description

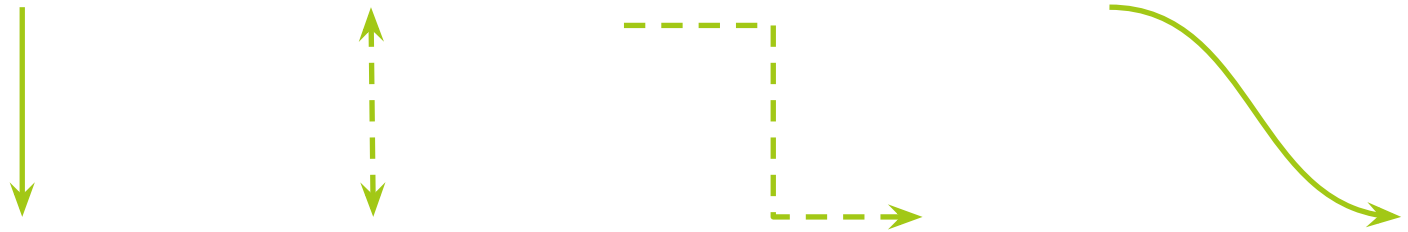
- A **flowchart** is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution to a given problem.

Basic Symbols

- **Start and end** symbols. Represented as circles, ovals or rounded rectangles, usually containing the word "Start" or "End", or another phrase signaling the start or end of a process.

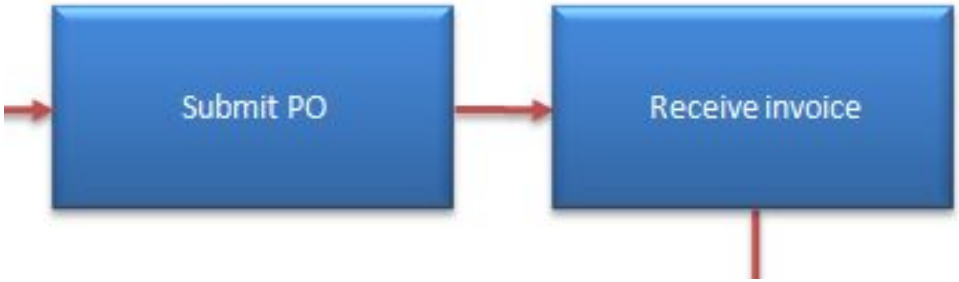
Basic Symbols

- **Arrows**. Showing "flow of control". An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to. The line for the arrow can be solid or dashed.



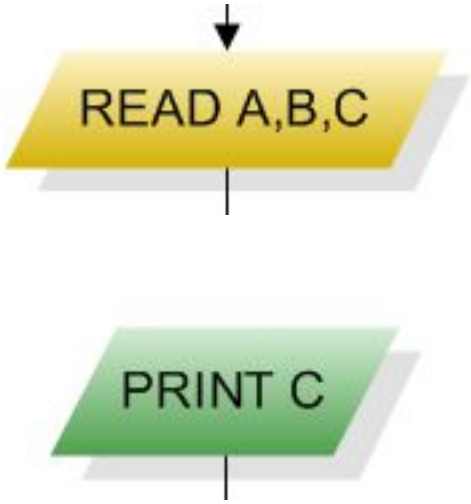
Basic Symbols

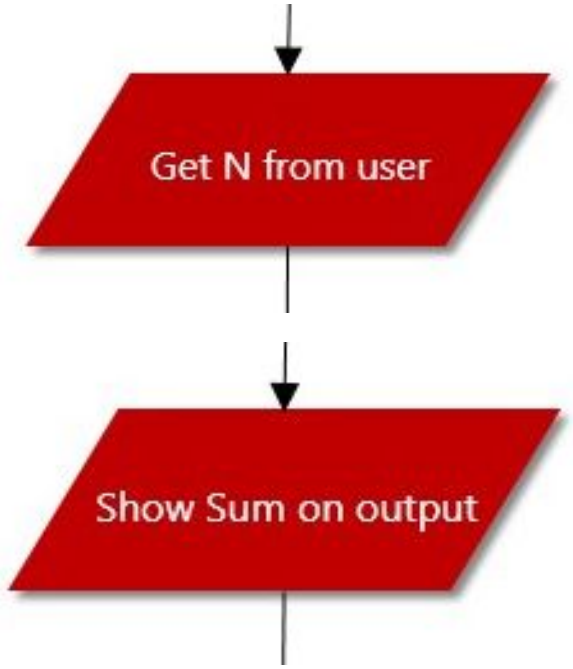
- **Generic processing steps** represented as rectangles.



Basic Symbols

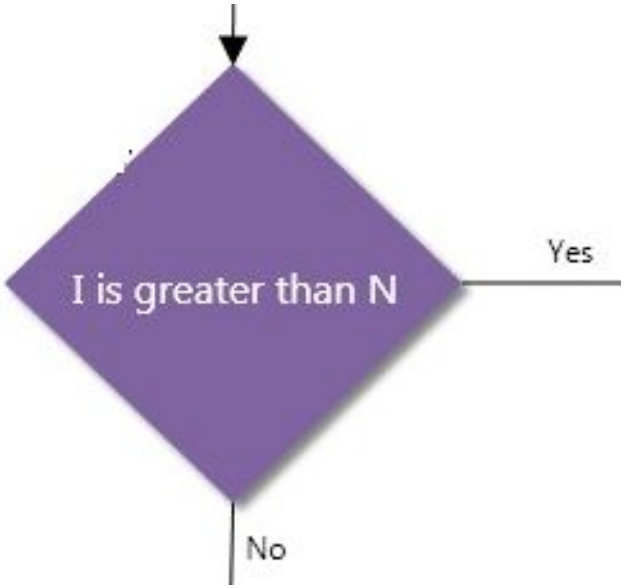
- **Input/Output** – represented as a parallelogram.

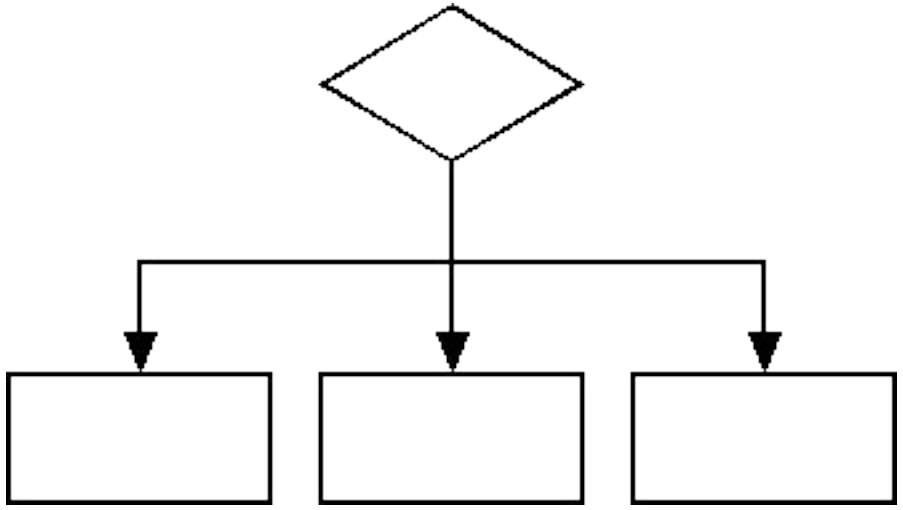




Basic Symbols

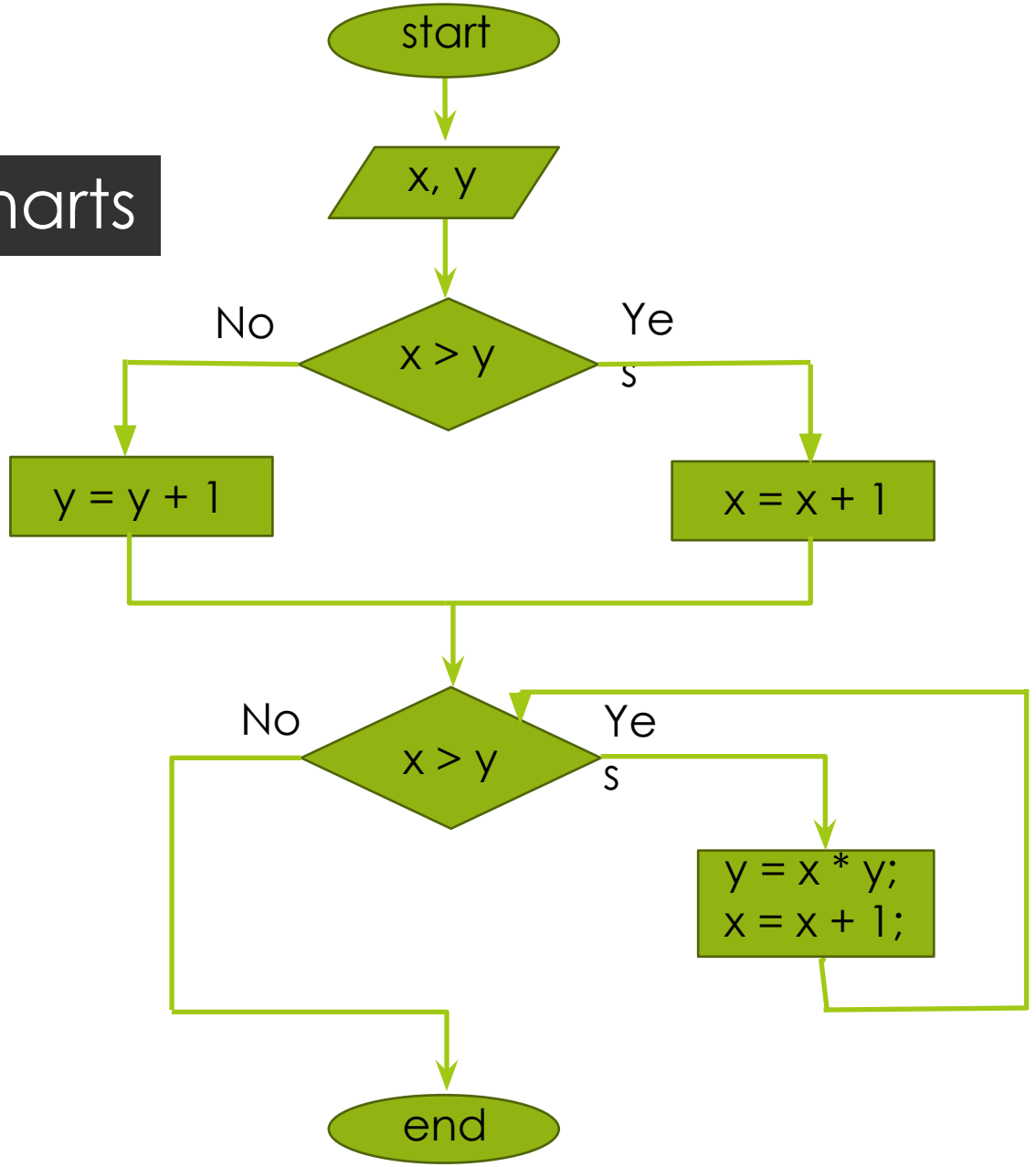
- **Conditional or decision** represented as a diamond (rhombus) showing where a decision is necessary, commonly a Yes/No question or True/False test.





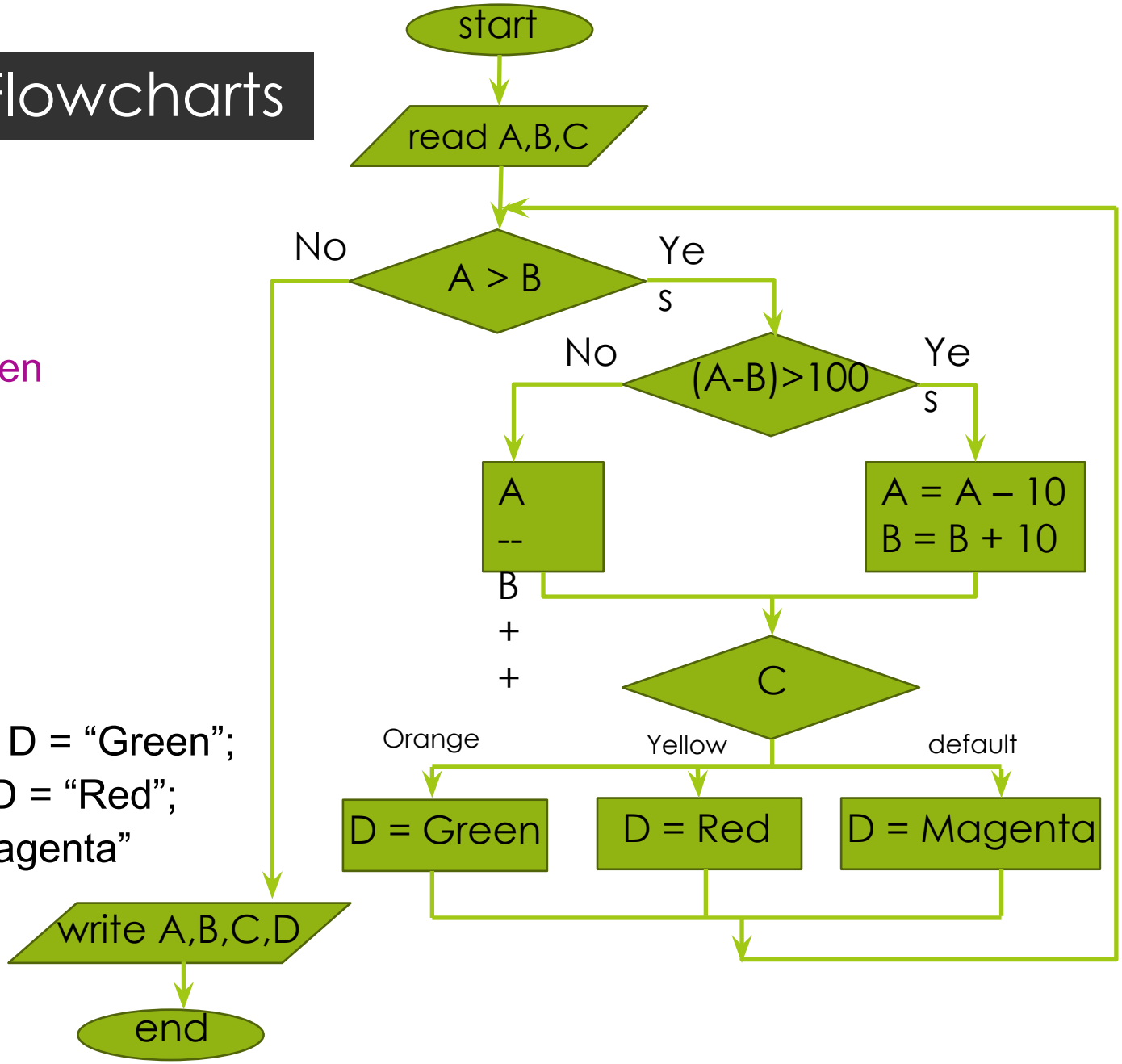
Practice 1. Flowcharts

```
read x;
read y;
if (x > y)
   x = x + 1;
else
   y = y + 1;
 while (x > y)
{
    y = x * y;
    x = x + 1;
}
```



Practice 2. Flowcharts

```
read A, B, C;
while (A > B) do
{
  if(A - B) > 100 then
      A = A - 10;
      B = B + 10
   else
      A--;
      B++;
  switch (C):
    case “Orange”: D = “Green”;
    case “Yellow”: D = “Red”;
    default: D = “Magenta”
}
write A, B, C, D;
```



White-box testing

Description

- **White-box testing** is testing that takes into account the internal mechanism of a system or component.
- White-box testing is also known as **structural testing**, **clear box testing**, and **glass box testing**.
- "Clear box" and "glass box" indicate that you have full visibility of the internal workings of the software product, specifically, the **logic** and the **structure of the code**.

White-box Testing Levels

- **Unit testing** – testing of individual hardware or software units or groups of related units. A **unit** is a software component that cannot be subdivided into other components.
- **Integration testing** – software components, hardware components, or both are combined and tested to evaluate the **interaction** between them.
- **System testing** – performed to analyze the behavior of the **whole system** according to the requirement specification. Business processes, system behavior, and system resources may also be taken into consideration. System testing is considered as the final test carried on the software from the development team.

Integration Testing Vs. System Testing

- **Integration testing** aims to check if the different sub functionalities or modules were **integrated properly** to form a bigger functionality. Focus of attention is on the modules. **Interface specifications** are taken into consideration.

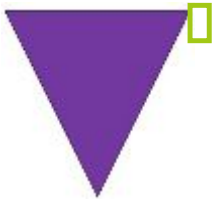
- In **system testing** the system is tested **as a whole**, where the functionalities that make up the system are not taken into consideration. The focus of attention is on system functionality. **Requirements specifications** are important.

- **Integration tests** are carried out **before** the system moves to the system testing level.

Integration Testing Approaches

- **Big Bang** - all or most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing.

- **Bottom Up Testing** is an approach where the lowest level components are tested first, then higher level components. The process is repeated until the component at the top of the hierarchy is tested.



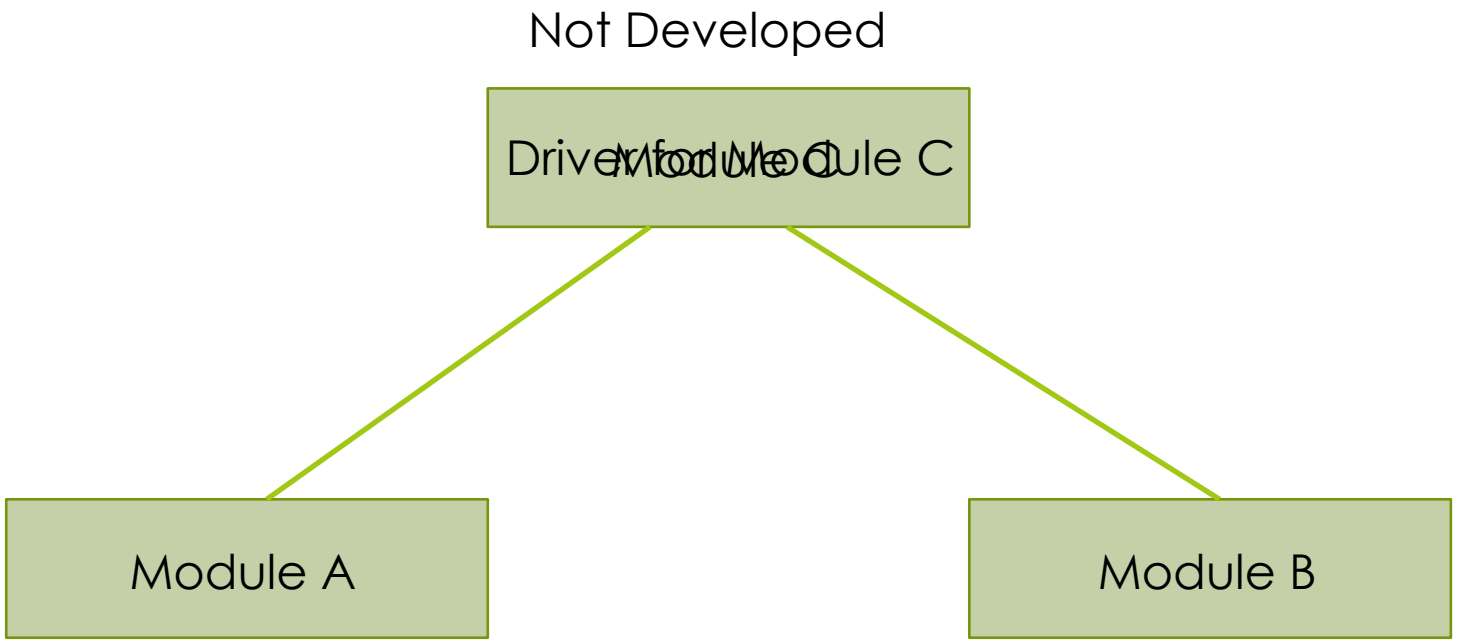
- **Top Down Testing** is an approach where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.

- **Sandwich Testing** is an approach to combine top down testing with bottom up testing.

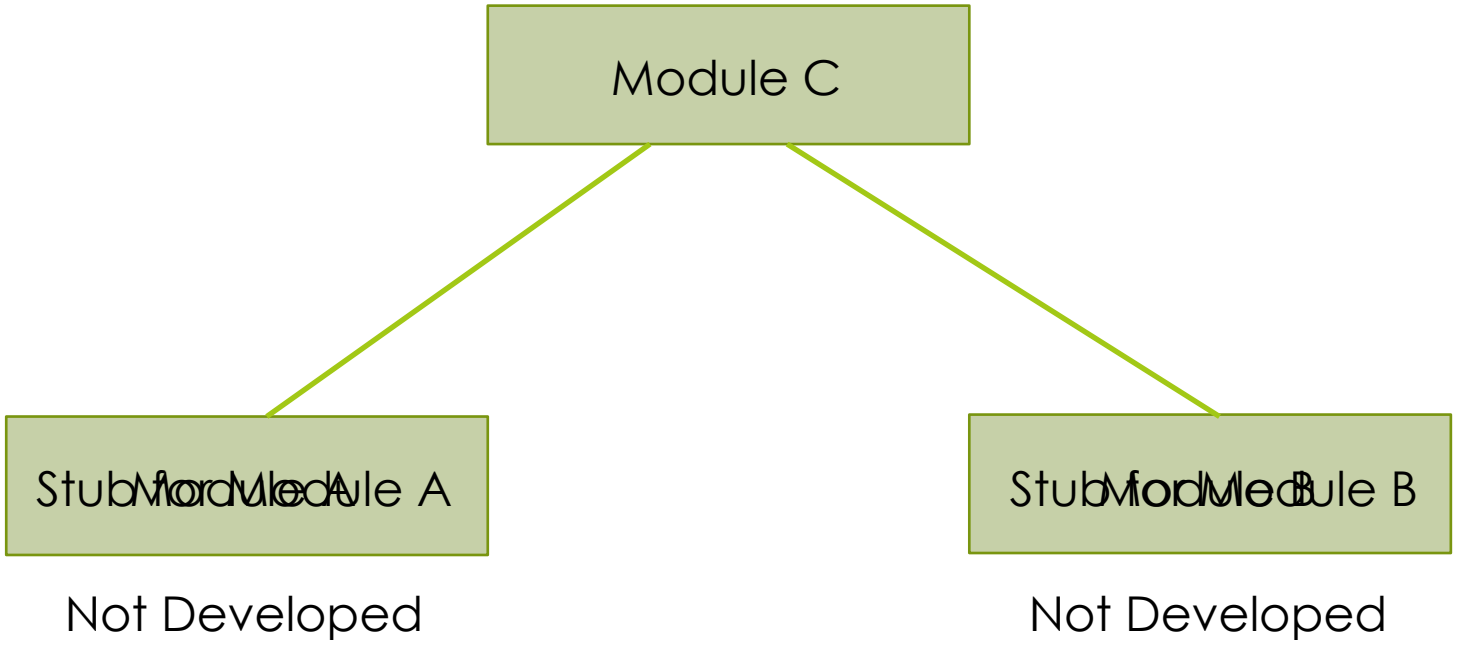
Stubs and Drivers

- A **driver (оболочка)** is a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results or most simplistically a line of code that calls a method and passes that method a value.
- A **stub (заглушка)** is a computer program statement substituting for the body of a software module that is or will be defined elsewhere or a dummy component or object used to simulate the behavior of a real component until that component has been developed.
- Stubs and drivers are often viewed as throwaway code. However, they do not have to be thrown away:
  - **Stubs** can be "filled in" to form the **actual method**.
  - **Drivers** can become **automated test cases**.

Stubs and Drivers. Example



Stubs and Drivers. Example



Static Testing

Static testing is a type of testing which requires only the source code of the product, not the binaries or executables. Static testing does not involve executing the programs on computers but involves people going through the code to find out whether:

- The code works according to the functional requirements.
- The code has been written in accordance with code conventions.
- The code for any functionality has been missed out.
- The code handles errors properly.

Static testing can be done by humans or with the help of specialized tools.

Cyclomatic Complexity

- **Cyclomatic complexity** (or conditional complexity) is a software metric. It was developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the **complexity of a program**. It is a measure of logical strength of the program. It directly measures the number of linearly independent paths through a program's source code.
- Cyclomatic complexity **more than 50** means **very high risks** and **non-testable code**.
- The complexity **M** is then defined as:

$$M = E - N + 2P,$$

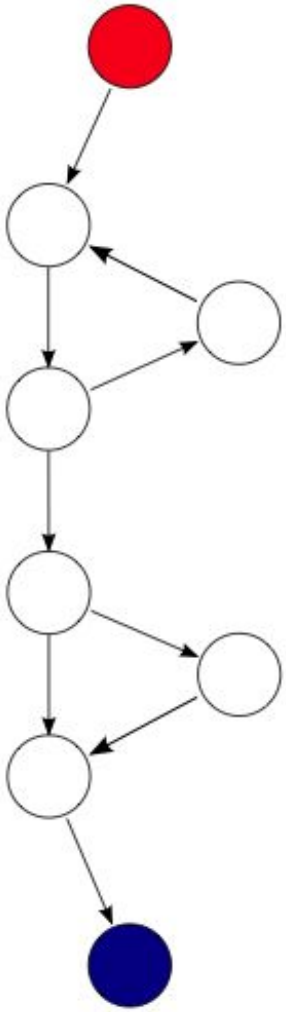
where:

E = the number of edges of the graph

N = the number of nodes of the graph

P = the number of connected components

Cyclomatic Complexity. Example



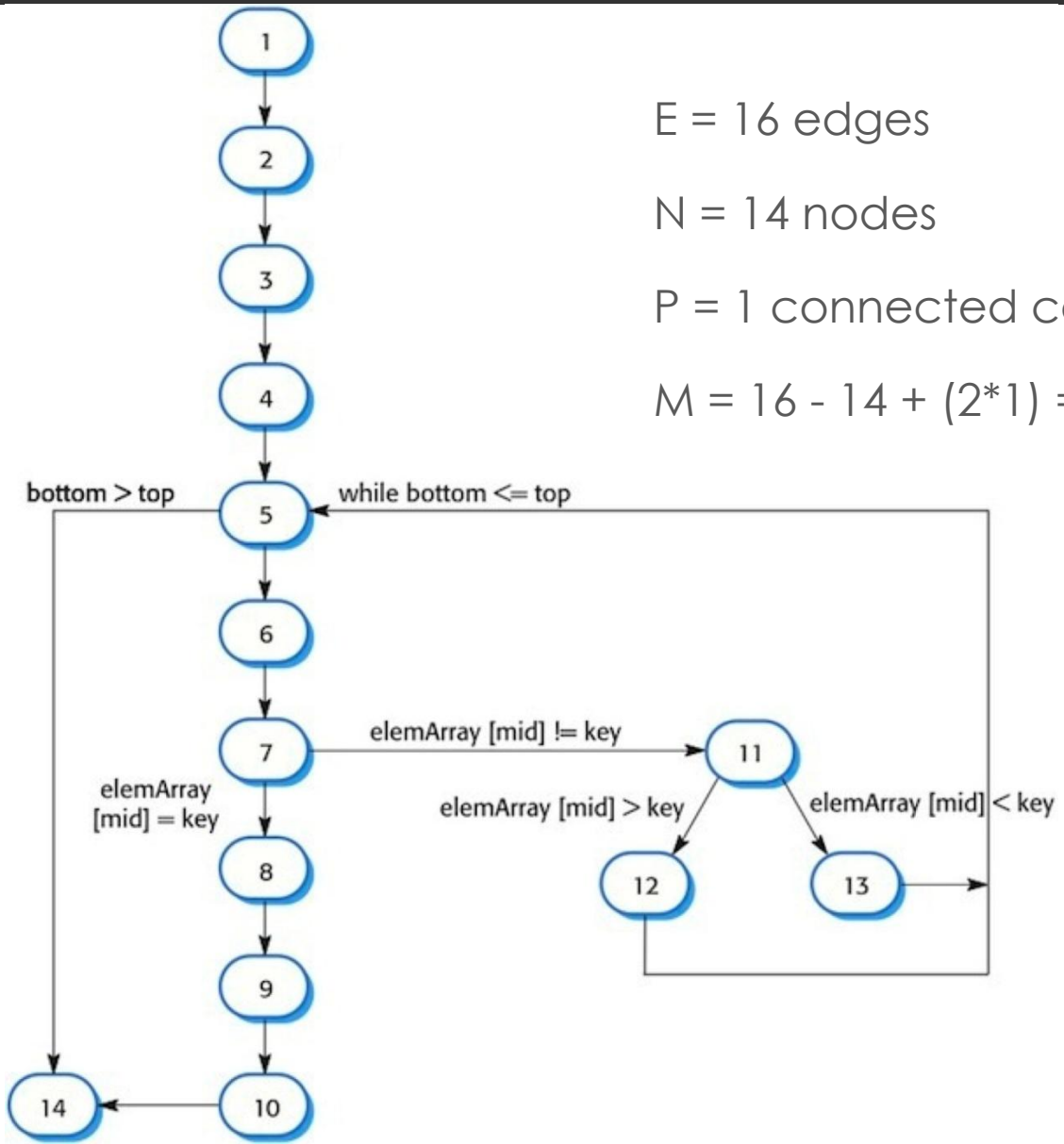
E = 9 edges

N = 8 nodes

P = 1 connected component

M = 9 - 8 + (2*1) = 3

Cyclomatic Complexity. Example



E = 16 edges

N = 14 nodes

P = 1 connected component

M = 16 - 14 + (2*1) = 4

Cyclomatic Complexity. Lifehack

- For programs **without goto** statements, the value of the cyclomatic complexity is **one more** than the **number of conditions** in the program.
- A **simple condition** is logical expression **without** ‘AND’ or ‘OR’ connectors.
- If the program **includes compound conditions**, which are logical expressions **including** ‘AND’ or ‘OR’ connectors, then you count the **number of simple conditions** in the compound conditions when calculating the cyclomatic complexity.
- `if (A < B)`
- `if ((A < B) && (C = false)) || (D > 506)`

Test Design Techniques

White-box test design techniques

- Control flow testing:
  - Statement testing
  - Branch testing*
  - Decision testing (Condition testing)*
  - Path testing
  - Multiple Conditions Testing
- Data flow testing
  - Define/use testing
  - “Program slices”

* - different meanings, but similar results

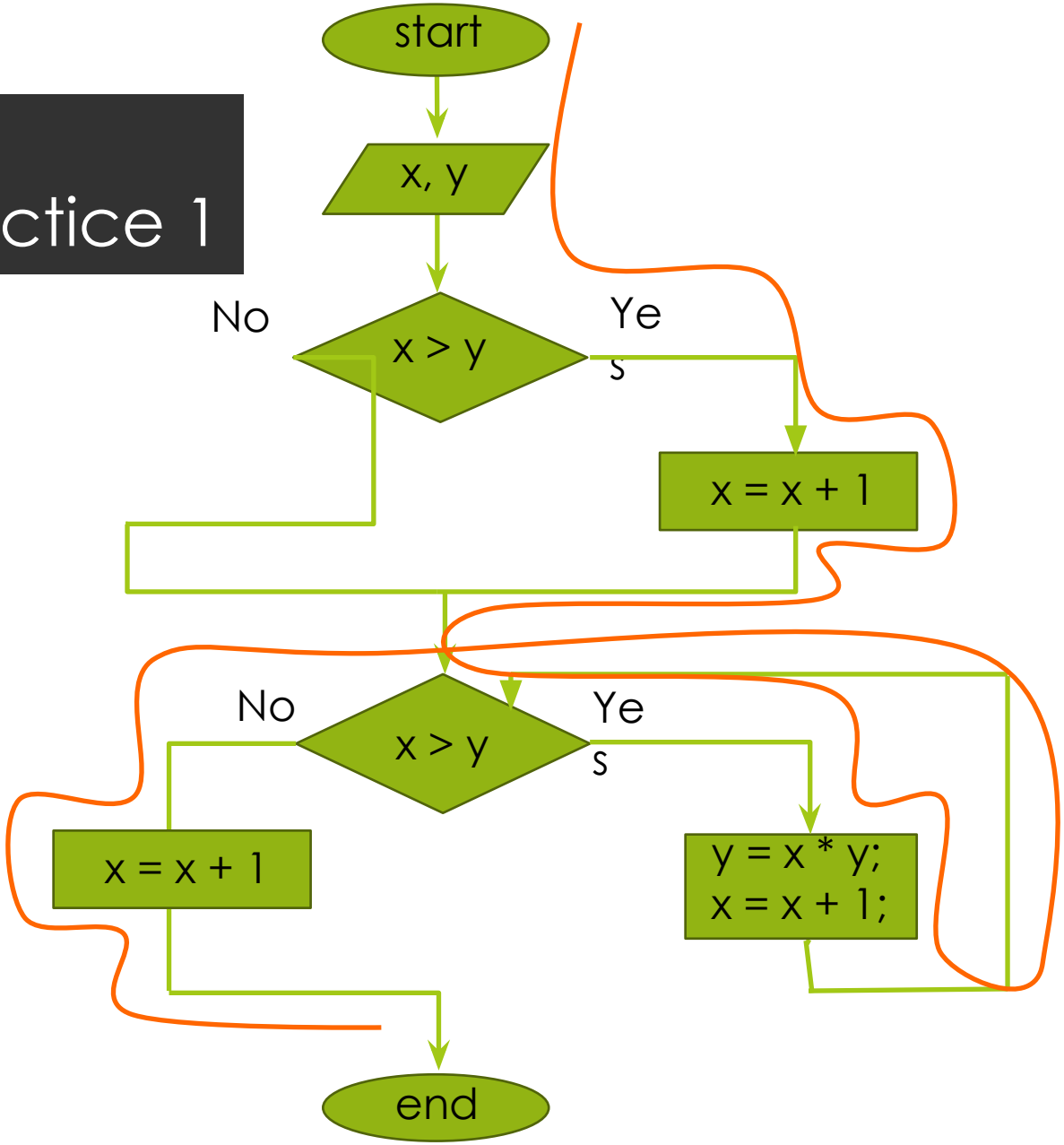
Control Flow Testing

- The starting point for control flow testing is a **program flow graph**. This is a skeletal model of all paths through the program.
- A flow graph consists of **nodes** representing **decisions** and **edges** showing **flow of control**.
- Each branch in a conditional statement (if-then-else or case) is shown as a separate path.

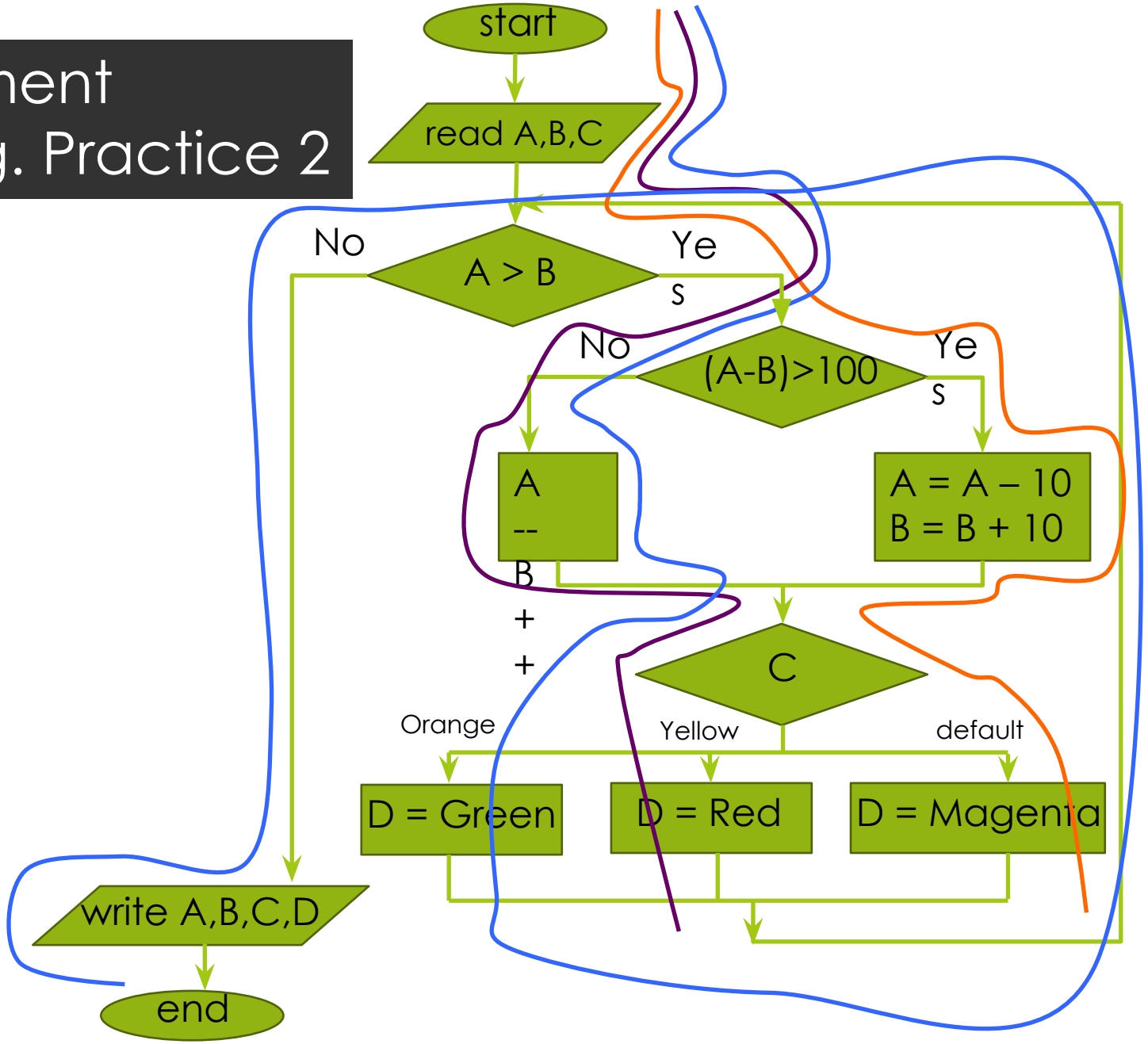
Statement Testing

- **Statement testing** – simply testing each statement. A statement exists as a single node within a program graph, and so if each node of the graph is traversed then so is each statement. Of course, this has the obvious problem that if a node has two edges, both of which will lead to every remaining node being traversed, this metric could be satisfied without every edge having been tested.
- Although it does not necessarily provide 100% coverage, the test metric is still widely accepted.
- **100% statement coverage doesn't guarantee 100% branch/decision or path coverage**.

Statement Testing. Practice 1



Statement Testing. Practice 2



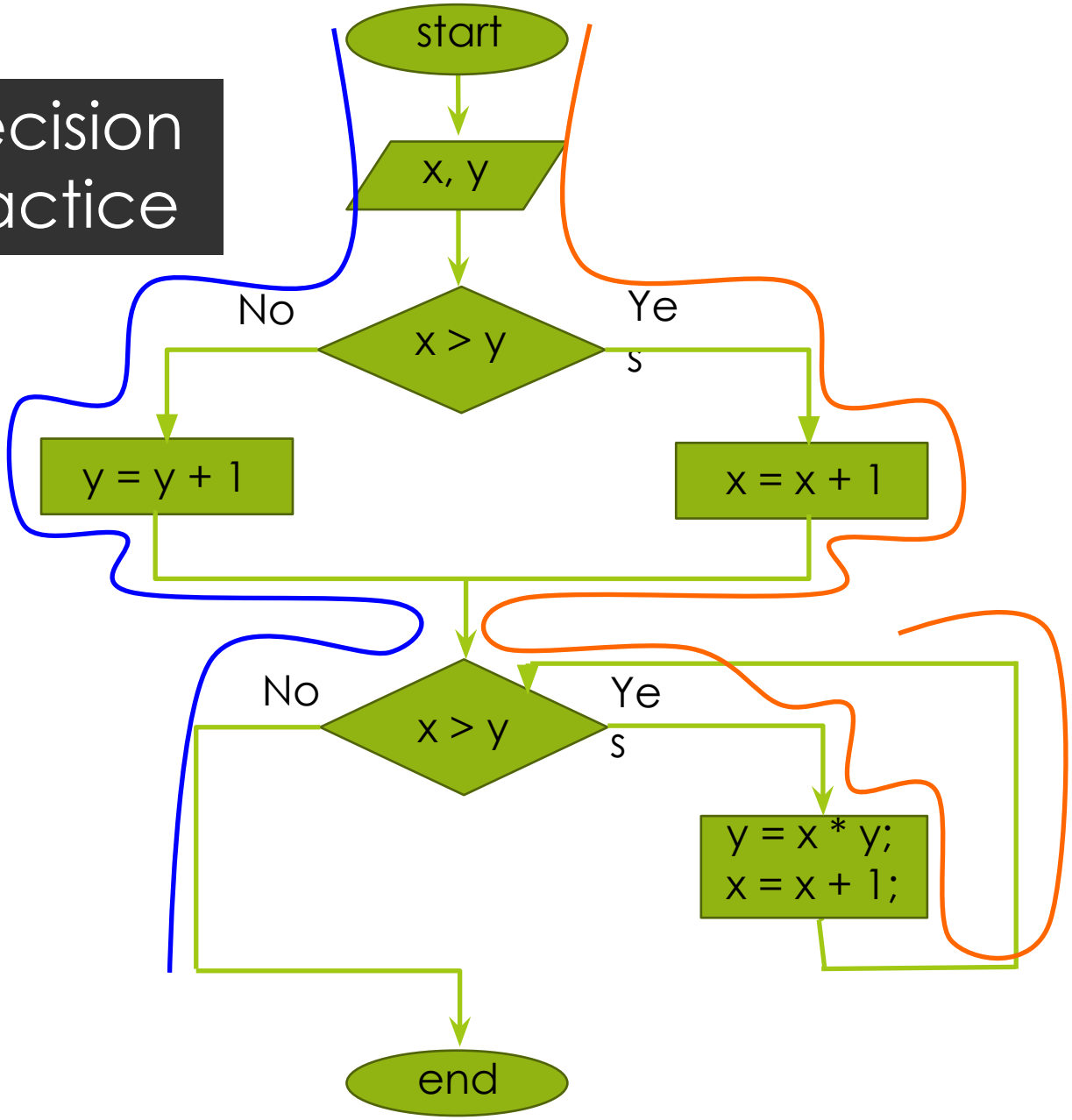
Branch/Decision Testing

- **Decision** – a program point at which the control flow has two or more alternatives. A node with two or more links to separate branches.
- **Decision testing** – test design technique in which test cases are designed to execute decision outcomes
- **Decision coverage** – percentage of decision outcomes that have been exercised by a test suite.
- **100% decision coverage implies both 100% branch coverage and 100% statement coverage**.

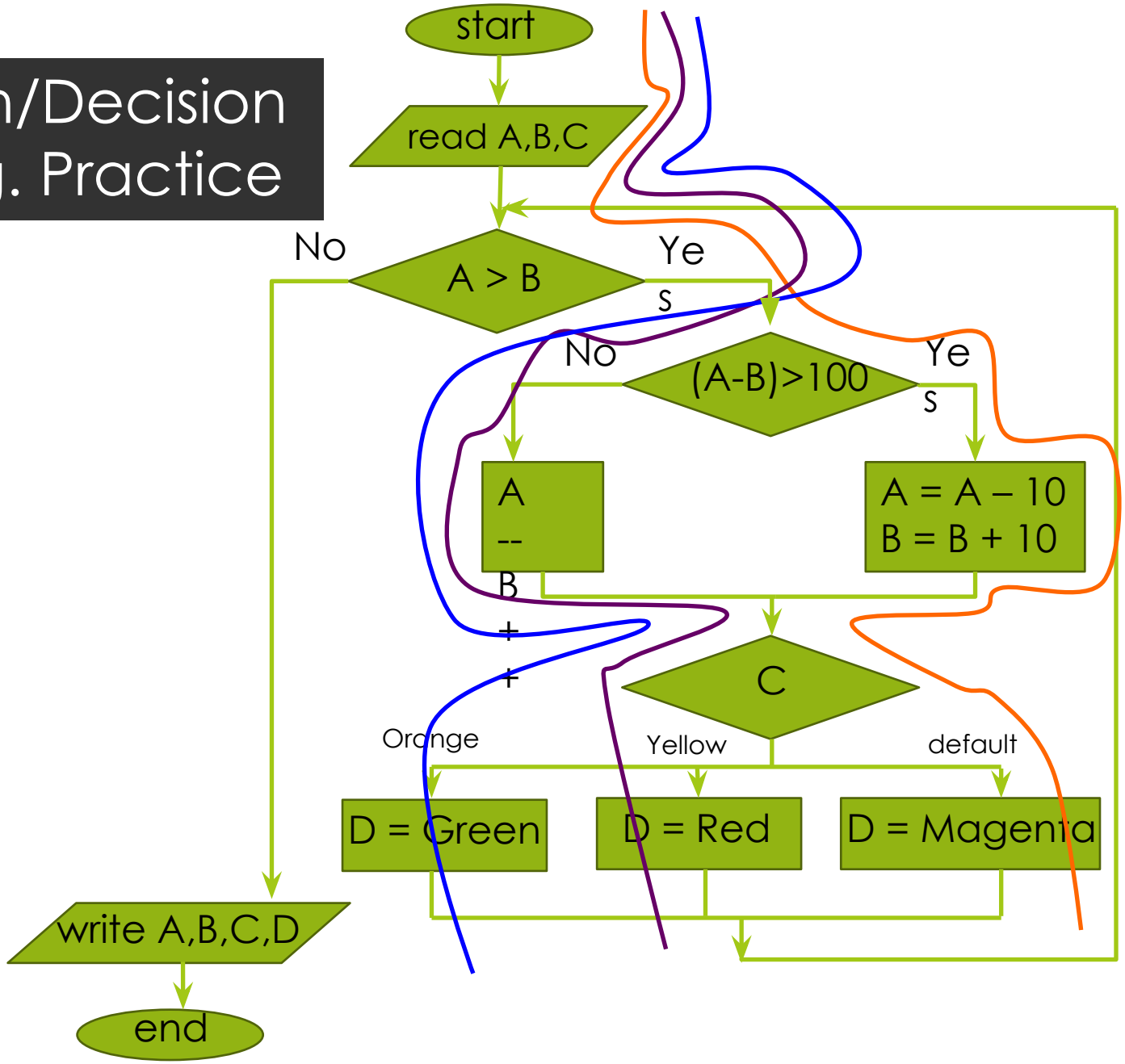
Branch/Decision Testing

- **A branch** – is the outcome of a decision.
- **Branch testing** – is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.
- **Branch coverage** – percentage of branches that have been exercised by a test suite. Simply measures which decision outcomes have been tested.
- **100% branch coverage implies both 100% decision coverage and 100% statement coverage.**

Branch/Decision Testing. Practice



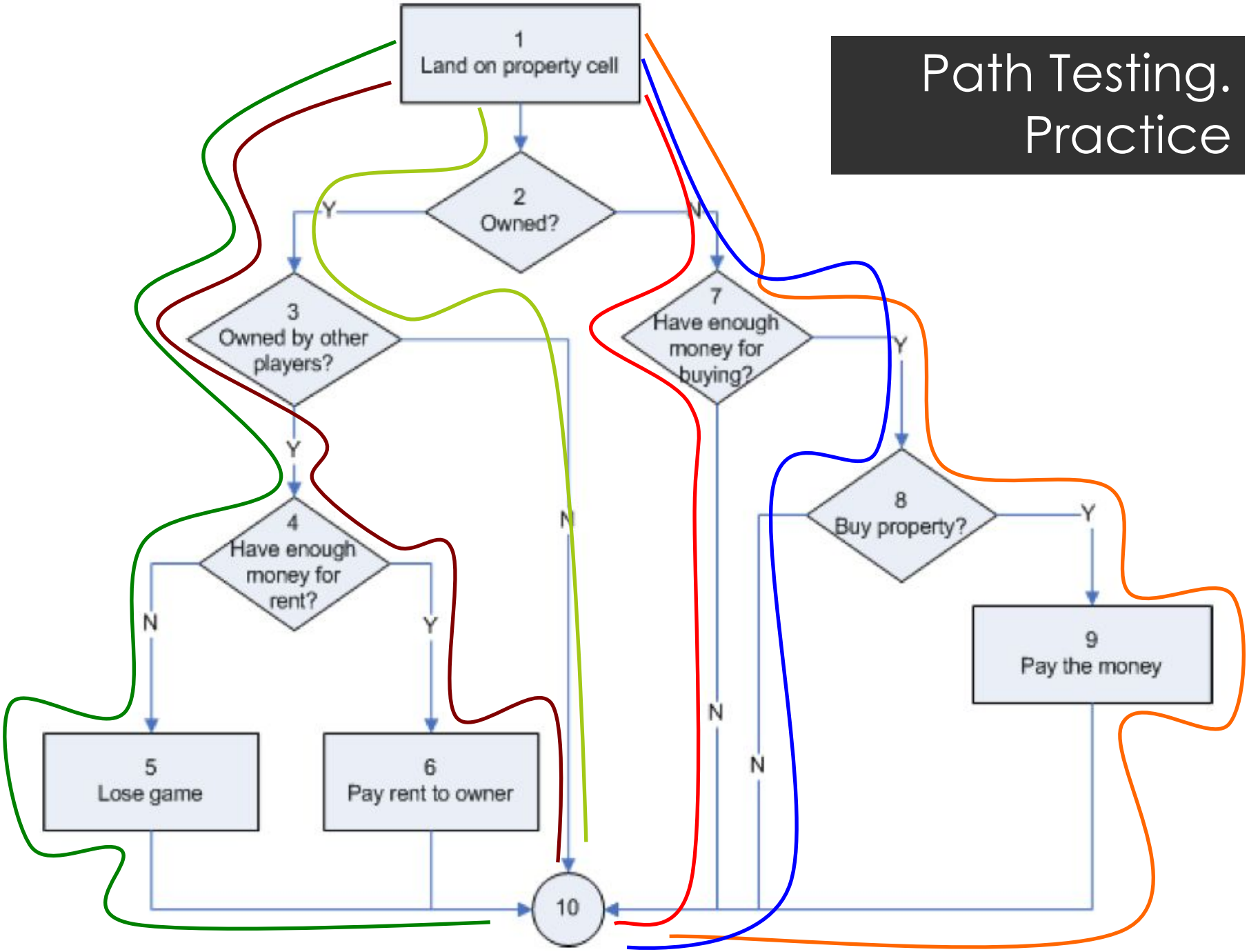
Branch/Decision Testing. Practice



Path Testing

- **Path testing** is a method for designing test cases intended to examine each possible linearly independent path of execution at least once.
- A **linearly independent path** is a sequence of commands without possible branch points.
- A **branch point** exists if a conditional permits alternative execution paths depending on the outcome of a logical test.
- **By creting tests for 100% path coverage, 100% statement and 100% branch/decision coverage can be guaranteed.**

Path Testing. Practice

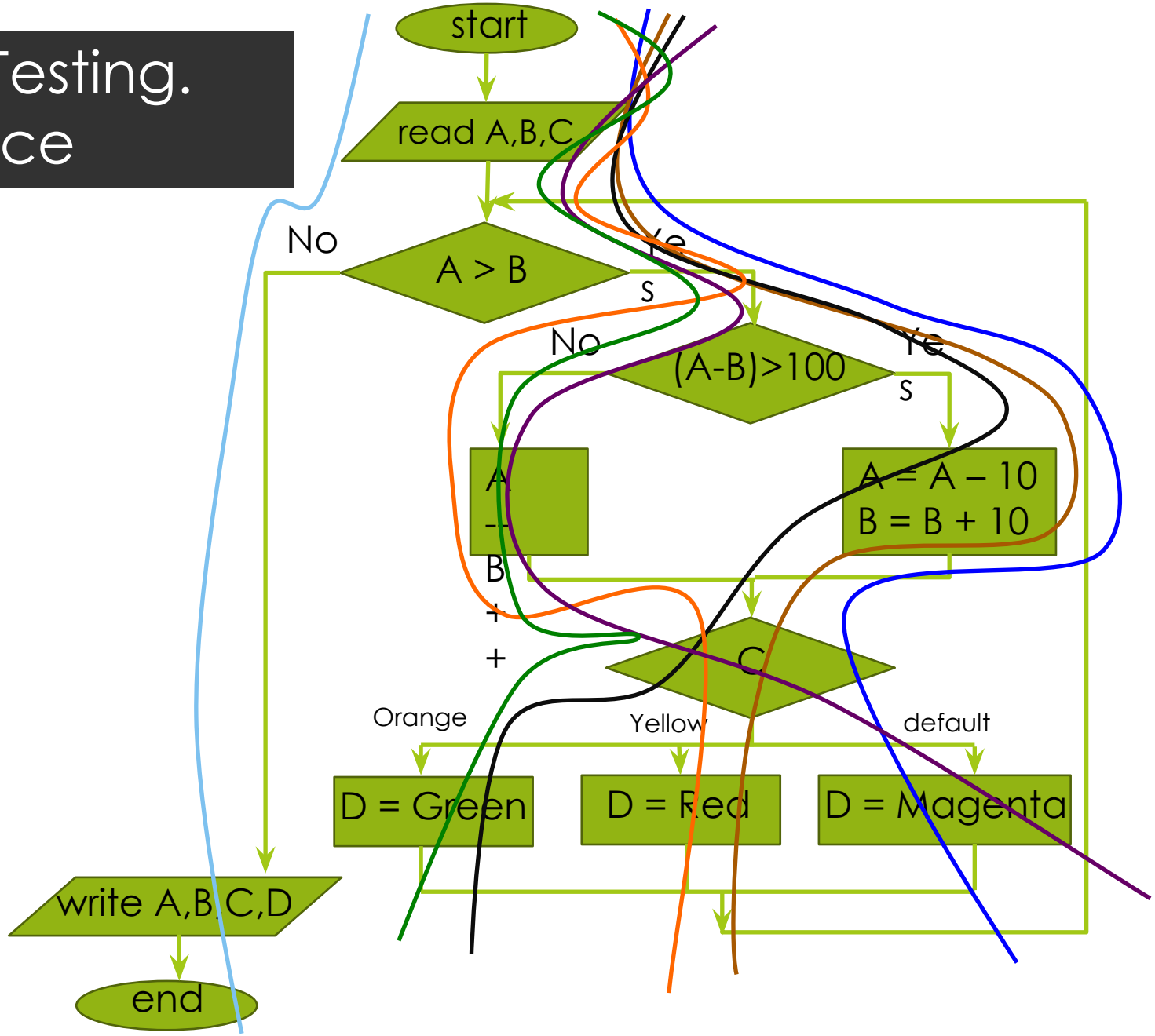


Minimum set of paths

- 1-2-3-4-5-10 (property owned by others, no money for rent)
- 1-2-3-4-6-10 (property owned by others, pay rent)
- 1-2-3-10 (property owned by the player)
- 1-2-7-10 (property available, don't have enough money)
- 1-2-7-8-10 (property available, have money, don't want to buy it)
- 1-2-7-8-9-10 (property available, have money, and buy it)

We would want to write a test case to ensure that each of these paths is tested at least once.

Path Testing. Practice



Multiple Conditions Testing

- **Multiple conditions testing** - a white-box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement).
- In Multiple Conditions Coverage for each decision **all the combinations** of conditions should be evaluated.

```
if (A || B) then
print C
```

The **test set** for Multiple Conditions Coverage will be:

A	B	Result
true	true	true
true	false	true
false	true	true
false	false	false

- **2^n** tests are needed, if there are **n conditions**.

Multiple Condition Testing. Example

```
if ((A < B) && (C == 158)) || (C > 506)
      I          II            III
```

I	II	III	A	B	C	Result
true	true	true	1	2	158/600	true
true	true	false	1	2	158	true
true	false	false	1	2	3	false
false	false	false	2	1	4	false
false	false	true	3	1	736	true
false	true	true	2	1	158/600	true
false	true	false	4	2	158	false
true	false	true	1	5	638	true

Complete coverage can never be achieved.

Memorize

- 100% Path coverage will imply 100% Statement coverage
- 100% Path coverage will imply 100% Branch/Decision coverage
- 100% Branch/Decision coverage will imply 100% Statement coverage
- Decision coverage includes branch coverage.
- **These rules work only such way – they don't work backwards.**

Fault Injection

Fault injection is a technique for improving the coverage of a test by introducing faults to test code paths. This includes:

- **Bebugging** (or fault seeding) is a software engineering technique to measure test coverage. Known bugs are randomly added to a program source code and the programmer is tasked to find them. The percentage of the known bugs not found gives an indication of the real bugs that remain.

- **Mutation testing** involves modifying a program's source code in small ways. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called killing the mutant. Test suites are measured by the percentage of mutants that they kill.

Practice

Practice. Example 1

How many test cases are necessary to cover all the possible sequences of statements for the following program fragment? Assume that the two conditions are independent of each other:

```
if (Condition 1) then
   statement 1
else statement 2
fi
if (Condition 2)then
   statement 3
fi
```

A. 3 Test Cases

B. 2 Test Cases

C. 4 Test Cases

D. Not achievable

Practice. Example 2

Given the following code, which is true:

```
if A > B then
    C = A – B
else
    C = A + B
endif
read D
if C = D then
    Print "Error"
endif
```

A. 1 test for statement coverage, 3 for branch coverage

B. 2 tests for statement coverage, 2 for branch coverage

C. 2 tests for statement coverage. 3 for branch coverage

D. 3 tests for statement coverage, 3 for branch coverage

E. 3 tests for statement coverage, 2 for branch coverage

Practice. Example 3

Given the following fragment of code, how many tests are required for 100% decision coverage? Please provide graph.

```
if width > length then
   biggest_dimension = width
        if height > width then
          biggest_dimension = height
        end_if
else
   biggest_dimension = length
   if height > length then
     biggest_dimension = height
   end_if
end_if
```

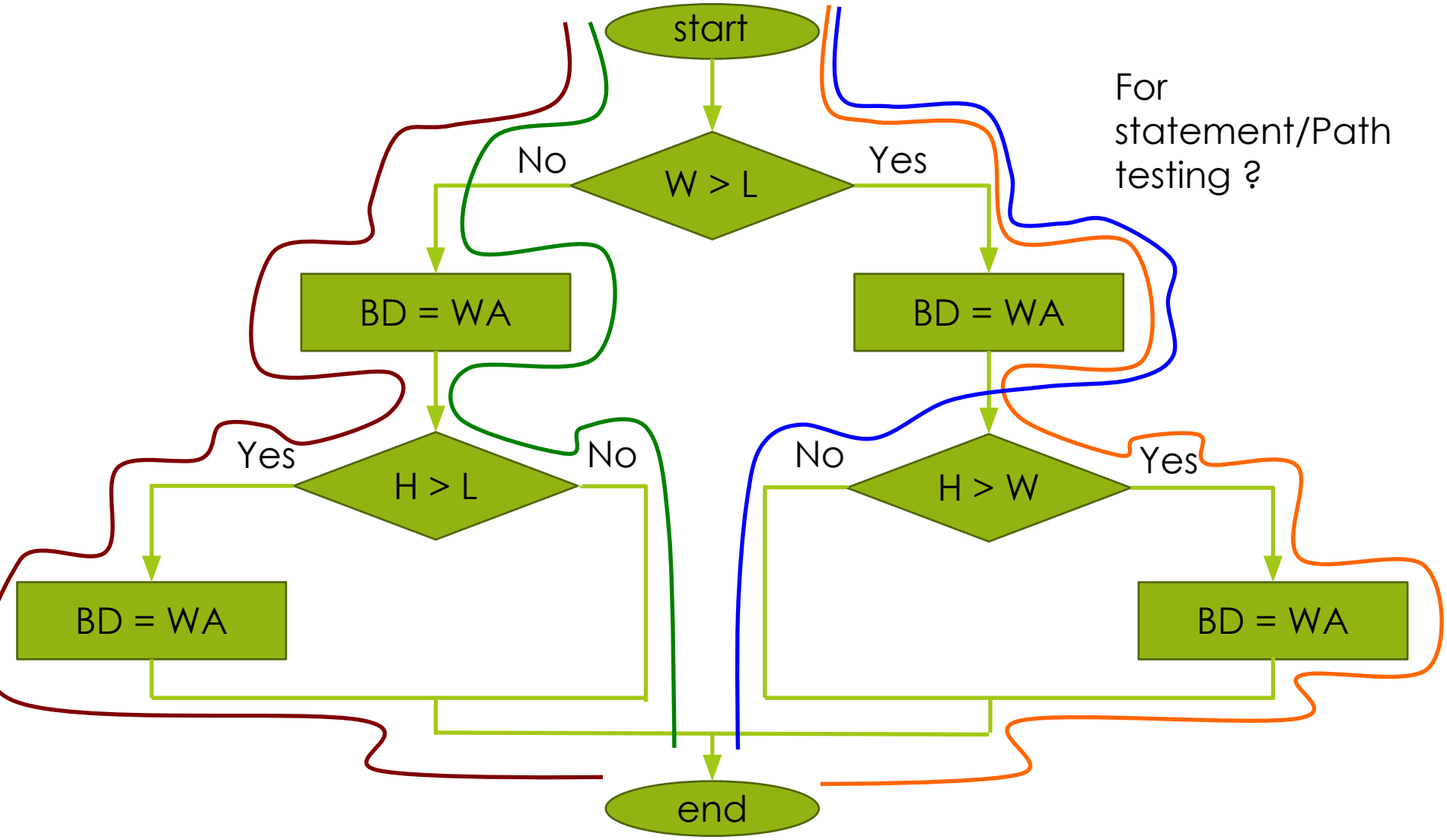
A. 3

B. 2

C. 1

D. 4

Graph for Example 3



Practice. Example 4

What is the smallest number of test cases required to provide 100% branch coverage?

```
if(x > y)
   x = x + 1;
else
   y = y + 1;
while(x > y)
{
   y = x * y;
   x = x + 1;
}
```

A. 1

B. 2

C. 3

D. 4

Practice. Example 5

Analyze the following highly simplified procedure:

Ask: "What type of ticket do you require, single or return?"
IF the customer wants 'return'
Ask: "What rate, Standard or Cheap-day?"
IF the customer replies 'Cheap-day'
Say: "That will be £11:20"
ELSE
Say: "That will be £19:50"
ENDIF
ELSE
Say: "That will be £9:75"
ENDIF

Now decide the minimum number of tests that are needed to ensure that all the questions have been asked, all combinations have occurred and all replies given.

A. 3
B. 4
C. 5
D. 6

Practice. Example 6

You have designed test cases to provide 100% statement and 100% decision coverage for the following fragment of code:

```
if width > length then
   biggest_dimension = width
else
   biggest_dimension = length
end_if
```

The following has been added to the bottom of the code fragment above:

```
print "Biggest dimension is " & biggest_dimension
print "Width: " & width
print "Length: " & length
```

How many more test cases are required?

A. One more test case will be required for 100 % decision coverage.

B. Two more test cases will be required for 100 % statement coverage, one of which will be used to provide 100% decision coverage.

C. None, existing test cases can be used.

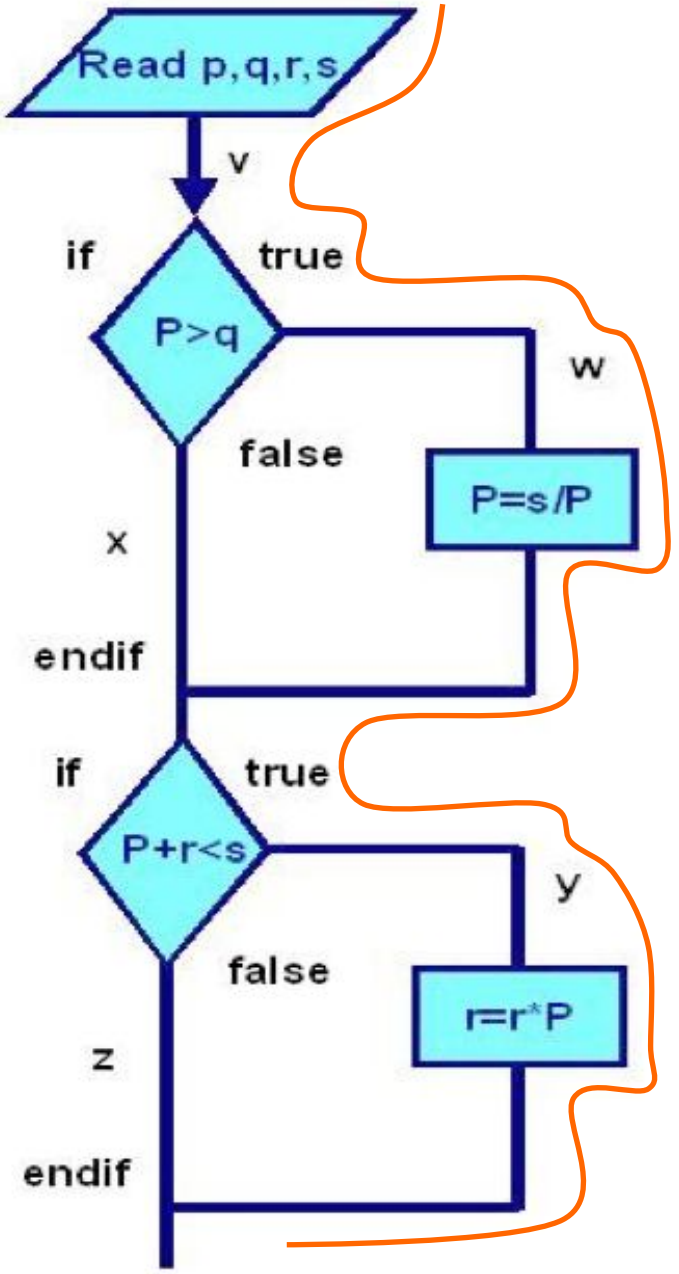
D. One more test case will be required for 100" statement coverage.

Practice. Example 7

The diagram represents the following paths through the code.

A. vwy
B. vwz
C. vxy
D. vxz

What is the MINIMUM combination of paths required to provide full **statement** coverage?



Practice. Example 8

Which of the following statements are correct?

- I.100% statement coverage guarantees 100% branch coverage.
- II.100% branch coverage guarantees 100% statement coverage.
- III.100% branch coverage guarantees 100% decision coverage.
- IV.100% decision coverage guarantees 100% branch coverage.
- V.100% statement coverage guarantees 100% decision coverage.

Practice. Example 9

If a program is tested and 100% branch coverage is achieved, which of the following coverage criteria is then guaranteed to be achieved?

- A. 100% Equivalence class coverage
- B. 100% Condition coverage and 100% Statement coverage
- C. 100% Statement coverage
- D. 100% Multiple condition coverage

Practice. Example 10

Which of the following statements is NOT correct?

- A. A minimal test set that achieves 100% decision coverage will also achieve 100% branch coverage.
- B. A minimal test set that achieves 100% path coverage will also achieve 100% statement coverage.
- C. A minimal test set that achieves 100% path coverage will generally detect more faults than one that achieves 100% statement coverage.
- D. A minimal test set that achieves 100% statement coverage will generally detect more faults than one that achieves 100% path coverage.

Data Flow Testing

Data Flow Testing. Description

- Variables are defined and used at different points within the program, the concept of **Data Flow Testing** allows the tester to **examine variables** throughout the program
- Data flow testing focuses on the variables used within a program.
- it is closely **related to path** testing, however the paths are selected on variables.

Data Flow Testing. Description

Data Flow testing helps to find such errors:

- A variable that is defined but never used (referenced).
- A variable that is used but never defined.
- A variable that is defined twice before it is used.

Data Flow Testing. Types

There are two major forms of data flow testing:

- Define/use testing
- “Program slices”

Staff Discount Program

- The owner of a shop has decided that her staff can have a 10 percent discount on all their purchases. If they spend more than £15, then the total discount is increased by 50 pence. The price of each item being purchased is input into the program. When -1 is entered, the total price is displayed, as well as the calculated discount and the final price to pay.

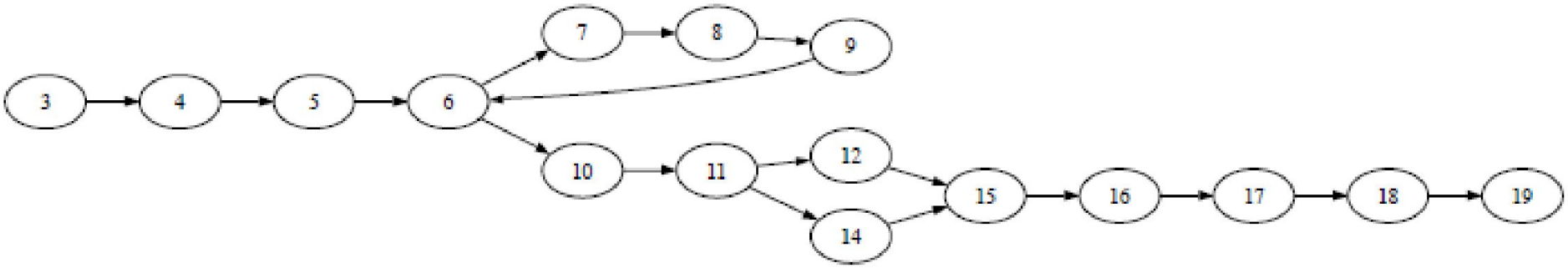
- For example, the values £5.50, £2.00 and £2.50 are input, equalling £10.00. The total discount would equal £1.00 (10% of £10.00), with the total price to pay equalling £9.00.

- A second example would have purchases of £10.50 and £5.00, equaling £15.50. In this case, as the total value is over £15, the discount would be £2.05 (10% of £15.50 is £1.55, plus 50p as the original total is over £15), meaning that the total price to pay would be £13.45.

Staff Discount Program

```
1   program Example()
2   var staffDiscount, totalPrice, finalPrice, discount, price
3   staffDiscount = 0.1
4   totalPrice = 0
5   input (price)
6   while (price != -1) do
7    totalPrice = totalPrice + price
8    input (price)
9   od
10  print ("Total price: " + totalPrice)
11  If (totalPrice > 15.00) then
12    discount = (staffDiscount * totalPrice) + 0.50
13  else
14    discount = staffDiscount * totalPrice
15  fi
16  print("Discount: " + discount)
17  finalPrice = totalPrice – discount
18  print("Final price: " + finalPrice)
19  endprogram
```

Graph for Staff Discount Program

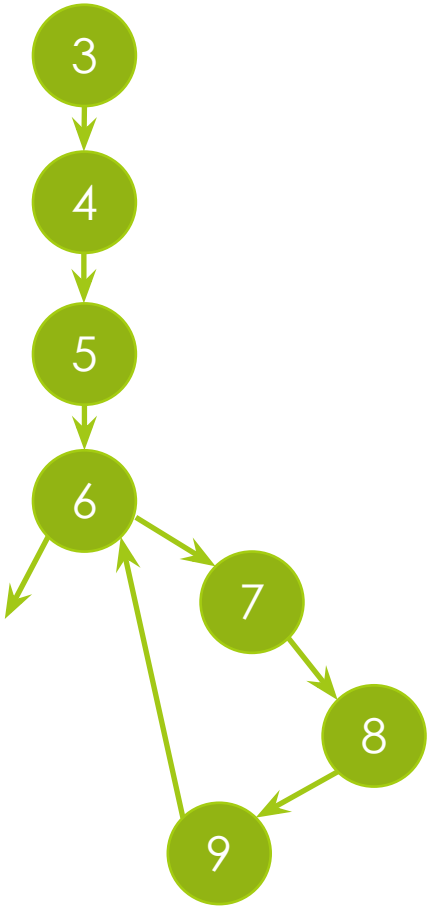


Each node in the graph corresponds to a statement in the program; however, lines 1 and 2 do not correspond to any node. This is because these lines are not used in the actual code of the program: they are used by the compiler to indicate the start of the program and to assign space in memory for the variables.

Definition/Use. Part 1

```
3 staffDiscount = 0.1
4 totalPrice = 0
5 input (price)
6 while (price != -1) do
7  totalPrice = totalPrice + price
8  input (price)
9 od
```

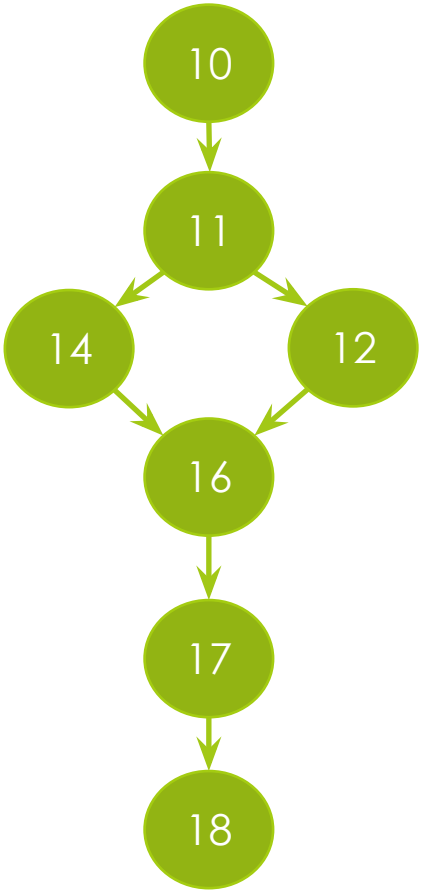
3 – definition of staffDiscount
4 – definition of totalPrice
5 – use of price
6 – use of price
7 – definition of totalPrice, use of totalPrice, use of price
8 – use of price



Definition/Use. Part 2

```
10 print ("Total price: " + totalPrice)
11 if (totalPrice > 15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print("Discount: " + discount)
17 finalPrice = totalPrice – discount
18 print("Final price: " + finalPrice)
```

10 – use of totalPrice
11 – use of totalPrice
12 – definition of discount, use of staffDiscount, totalPrice
14 - definition of discount, use of staffDiscount, totalPrice
16 – use of discount
17 – definition of finalPrice, use of totalPrice, discount
18 – use of finalPrice



Define/Use Testing

- "**Define/Use**" refers to the two main aspects of a variable: it is either **defined** (a value is assigned to it) or **used** (the value assigned to the variable is used elsewhere – maybe when defining another variable).
- The program is called **P**
- Its graph as **G(P)**. The program graph has single entry and exit nodes, and there are no edges from a node to itself.
- The set of variables within the program is called **V**

Defining/Usage Nodes

- Within the context of define/use testing there are two types of nodes.
- **Defining nodes, referred to as DEF(v, n):** Node n in the program graph of P is a **defining node** of a variable v in the set V if and only if **at n, v is defined**. For example, with respect to a variable x, nodes containing statements such as "input x" and "x = 2" would both be defining nodes.
- **Usage nodes, referred to as USE(v, n):** Node n in the program graph of P is a **usage node** of a variable v in the set V if and only if **at n, v is used**. For example, with respect to a variable x, nodes containing statements such as "print x" and "a = 2 + x" would both be usage nodes.

Usage nodes. Types

The two major types of usage nodes are:

- **P-use:** predicate use – the variable is used when making a decision (e.g. if b > 6).
- **C-use:** computation use – the variable is used in a computation (for example, b = 3 + d – with respect to the variable d).
  - **O-use:** output use – the value of the variable is output to the external environment (for instance, the screen or a printer).
  - **L-use:** location use – the value of the variable is used, for instance, to determine which position of an array is used (e.g. a[b]).
  - **I-use:** iteration use – the value of the variable is used to control the number of iterations made by a loop (for example: for (int i = 0; i <= 10; i++)).

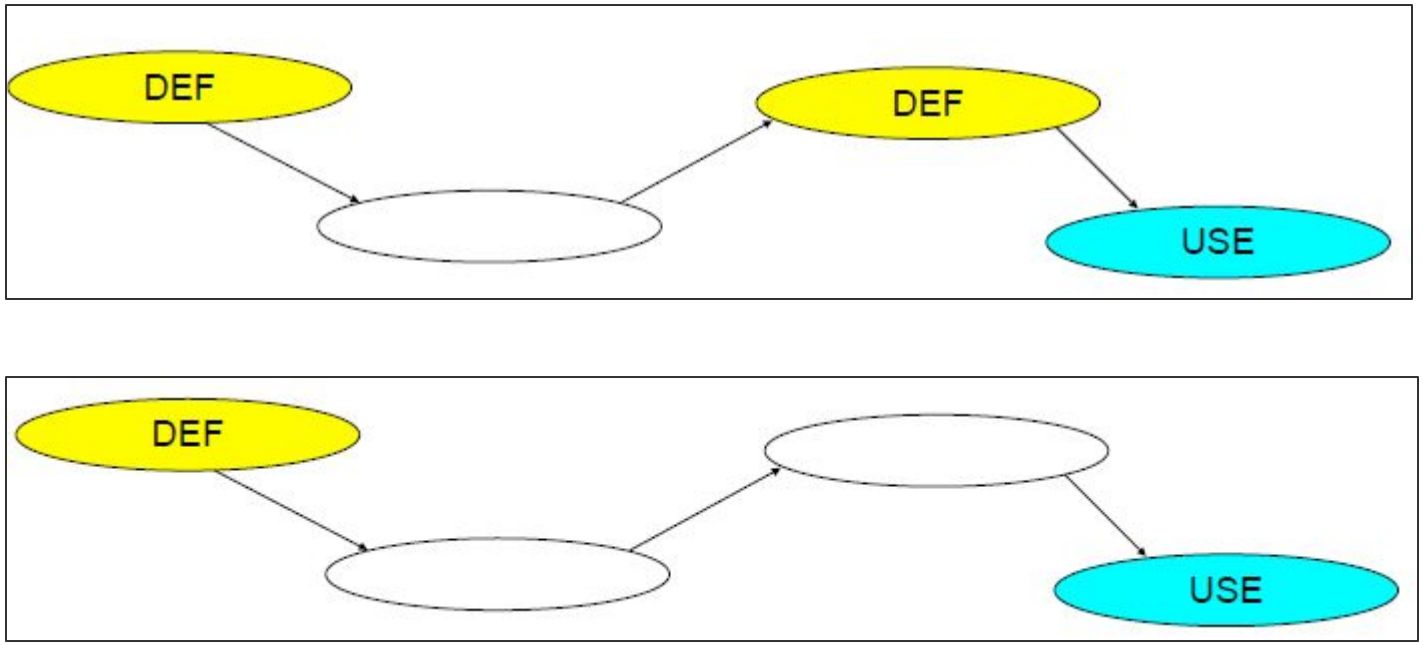
The defining and usage nodes for the variable totalPrice

Node	Type	Code
4	DEF	**totalPrice** = 0
7	DEF	**totalPrice** = totalPrice + price
7	USE	totalPrice = **totalPrice** + price
10	USE	print("Total price: " + **totalPrice**)
11	USE	if(**totalPrice** > 15.00) then
12	USE	discount = (staffDiscount * **totalPrice**) + 0.50
14	USE	discount = staffDiscount * **totalPrice**
17	USE	finalPrice = **totalPrice** - discount

Path Types

- **Definition-use (du) paths:** A path in the set of all paths in P(G) is a du-path for some variable v if and only if there exist DEF(v, m) and USE(v, n) nodes such that m is the first node of the path, and n is the last node.
- **Definition-clear (dc) paths:** A path in the set of all paths in P(G) is a dc-path for some variable v if and only if it is a du-path and the initial node of the path is the **only** defining node of v in the path.

DU/DC Paths



- First figure shows an example of a du-path. However, this path is not definition-clear, as there is a second defining node within the path.
- Second graph shows definition-clear path.

DU/DC Paths for Staff Discount Program

Looking at the Staff Discount Program, for the price variable there are two defining nodes and two usage nodes, as listed below:

- Defining nodes:
  - DEF(price, 5)
  - DEF(price, 8)
- Usage nodes:
  - USE(price, 6)
  - USE(price, 7)

Therefore, there are four du-paths:

- <5, 6>
- <5, 6, 7>
- <8, 9, 6>
- <8, 9, 6, 7>

All of these paths are definition-clear, so they are all dc-ps.

Coverage Metrics

- The set of paths satisfies All-Defs for P if within the set of v paths, every defining node for each variable in the program has a definition-clear path to a usage node for the same variable, within the set of paths chosen.
- The set of paths satisfies All-P-Uses for P if, within the set of paths, every defining node for each variable in the program has a definition-clear path to every P-use node for the same variable.
- The set of paths satisfies All P-Uses/Some C-Uses for P if, within the set of paths, every defining node for each variable in the program has a definition-clear path to every P-use node for the same variable: however, if there are no reachable P-uses, the definition-clear path leads to at least one C-use of the variable.
- The set of paths satisfies All C-Uses/Some P-Uses for P if, within the set of paths, every defining node for each variable in the program has a definition-clear path to every C-use node for the same variable: however, if there are no reachable C-uses, the definition-clear path leads to at least one P-use of the variable.
- The set of paths satisfies All-Uses for P if, within the set of paths, every defining node for each variable in the program has a definition-clear path to every usage node for the same variable.
- The set of paths satisfies All-DU-Paths for P if, the set of paths contains every feasible DU-path for the program.

“Program Slices”

- A **program slice** with respect to a variable at a certain point in the program, is the set of program statements from which the value of the variable at that point of the program is calculated.
- Program slices use the notation **S**(**V**, **n**), where **S** indicates that it is a program slice, **V** is the set of variables of the slice and **n** refers to the statement number (i.e. the node number with respect to the program graph) of the slice.
- The program slice allows the programmer to focus specifically on the code that is relevant to a particular variable at a certain point.

Slice for Staff Discount Program

So, for example, with respect to the price variable in Staff Discount Program, the following slices for each use of the variable can be created:

- S(price, 5) = {5}
- S(price, 6) = {5, 6, 8, 9}
- S(price, 7) = {5, 6, 8, 9}
- S(price, 8) = {8}

Conclusion

How Much Testing is Enough?

- Time
- Money
- People
- Requirements
- Knowledge
- Product
- Code

Pros and Cons

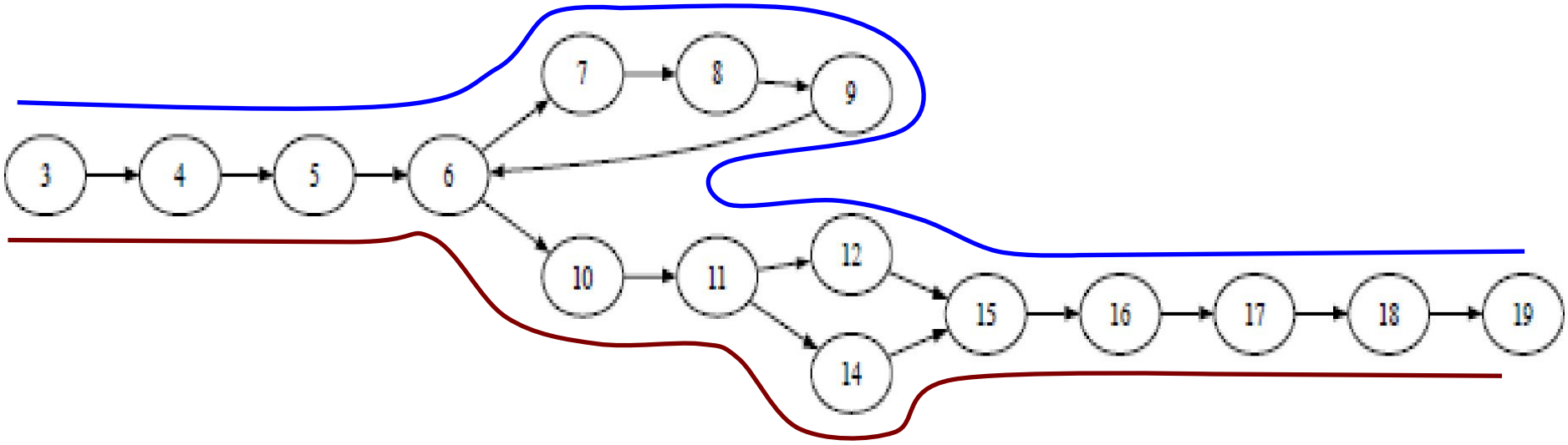
- White-box testing finds defects which hardly can be found within black-box testing.
- You can be absolutely sure, that your code is amazing.
- Advanced knowledge of source code.
- Developers carefully implement any new implementation.
- White-box testing doesn't take requirements into account.
- White-box testing is expensive and time consuming.
- White-box testing requires developer with advanced knowledge
- It is not realistic to test every single existing condition of the application.

Staff Discount Program

- The owner of a shop has decided that her staff can have a 10 percent discount on all their purchases. If they spend more than £15, then the total discount is increased by 50 pence. The price of each item being purchased is input into the program. When -1 is entered, the total price is displayed, as well as the calculated discount and the final price to pay.

- For example, the values £5.50, £2.00 and £2.50 are input, equalling £10.00. The total discount would equal £1.00 (10% of £10.00), with the total price to pay equalling £9.00.

- A second example would have purchases of £10.50 and £5.00, equaling £15.50. In this case, as the total value is over £15, the discount would be £2.05 (10% of £15.50 is £1.55, plus 50p as the original total is over £15), meaning that the total price to pay would be £13.45.


3
4
5
6
7
8
9
10
11
12
14
15
16
17
18
19


ANY
QUESTIONS
?

References

- Google ☺
- Software Testing and Continuous Quality Improvement by William E. Lewis
- Software Testing Principles and Practices by Srinivasan Desikan
- Software testing and quality assurance. Theory and practice by Kshirasagar Naik and Priyadarshi Tripathy
- http://books.google.com.ua/books?id=Yt2yRW6du9wC&printsec=frontcover&hl=ru#v=onepage&q&f=false
- ISTQB Syllabus
- White-box testing Techniques (Attached)
- Data Flow Testing (Attached)

/hite Box Testing Techniques

)ata Flow Testing

Thanks to

- Yanina Hladkova
- Andriy Yudenko
- Ivan Dmitrina


Fin