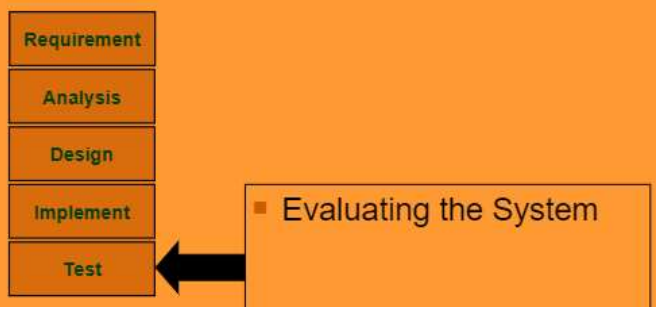
L13

Tuesday, April 11, 2023 11:02 PM

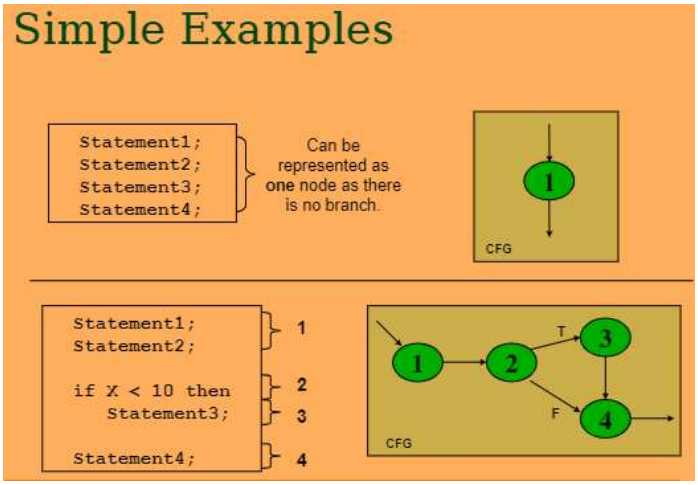
Overview of This Lecture

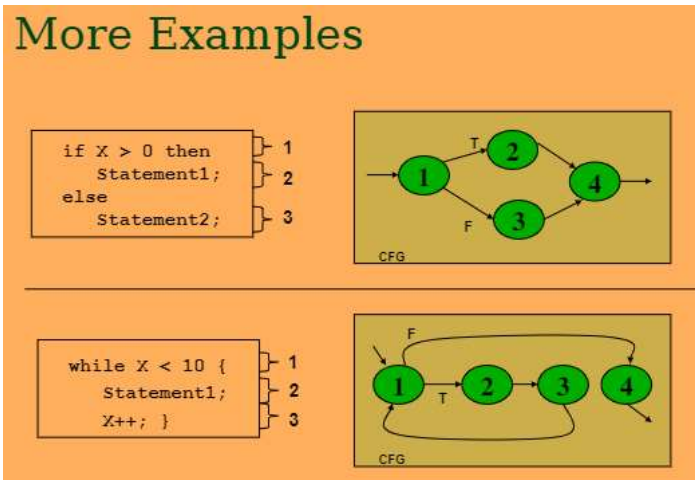
- Test Case Design
  - White Box
    - Control Flow Graph
    - Cyclomatic Complexity
    - Basic Path Testing
  - Black Box
    - Equivalence Classes
    - Boundary Value Analysis

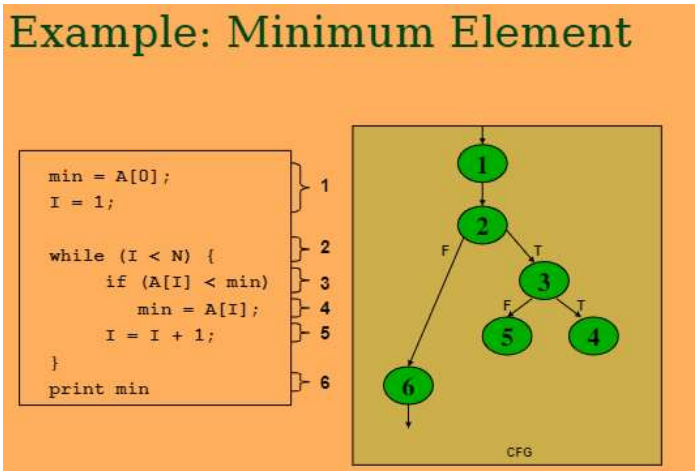
Where are we now?



- Test Case Design: Review
  - White box testing:
    - Knowing the internal workings of a component
    - Test cases exercise specific sets of condition, loops, etc.
  - Black box testing
    - Knowing the specified fn a component has been designed for
    - Test conducted at interface of the component
- White Box Testing: Intro
  - Test engineers have access to src code
  - Usually @ the Unit Test lvl as the pgrmers have knowledge of the internal logic of code
  - Tests based on coverage of
    - Code statements
    - Branches
    - Paths
    - Conditions
  - Most of the testing techniques are based on Control Flow Graph (CFG) of a code fragment
- Control Flow Graph: Intro
  - Abstract rep of a structured pgrm/fn/method
  - Has 2 main pts
    - Node - Reps a stretch of sequential code statements w/ no branches
    - Directed Edge – aka arc, reps a branch, alt path in execution
  - Path
    - Collection of Nodes linked w/ Directed Edges

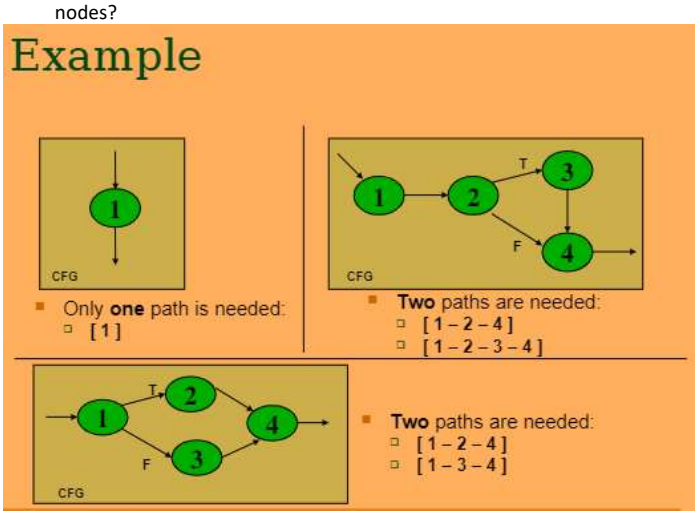
- 

- 

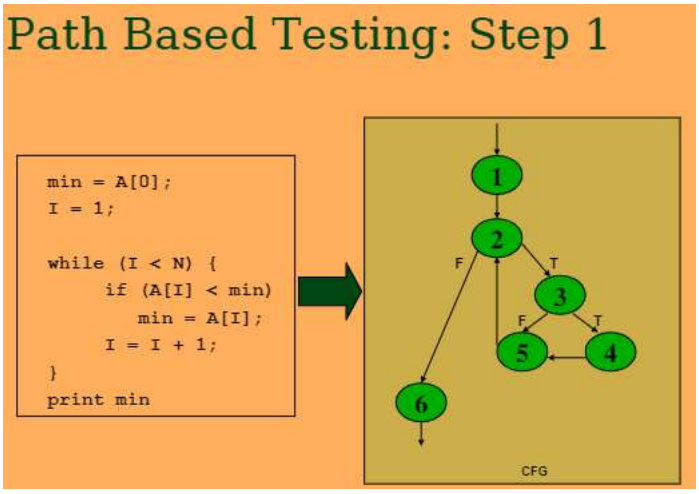
- Notation Guide for CFG (ctrl flow graph, not context free grammar lol)
  - CFG should have
    - 1 entry arc (aka directed edge)
    - 1 exit arc
  - All nodes should have
    - At least 1 entry arc
    - At least 1 exit arc
  - A **logical node** that doesn't rep any actual statements can be added as a joining pt for several incoming edges
    - Reps a logical closure
    - Ex: node 4 in the if-then-else ex
- 

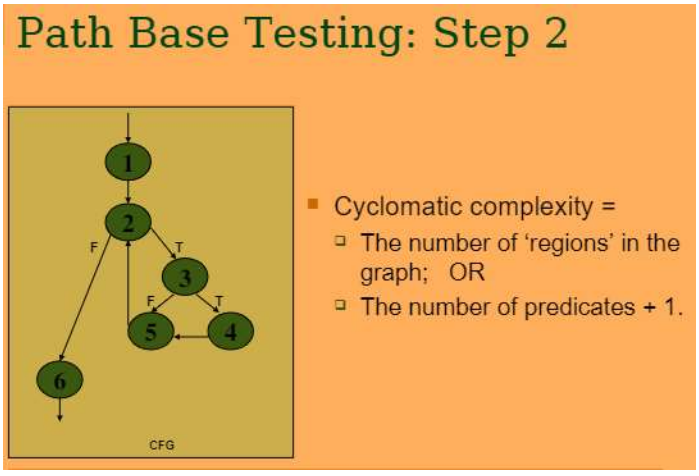
- Number of Paths thru CFG
  - Q: given a pgrm, how to exercise all statements and branches at least once?
  - Q: (translating a pgrm into a CFG, good question is) Given a CFG, how to cover all arcs and nodes at least once

- Since path is trail of nodes linked by arcs, its like asking: given a CFG, what is set of paths that can cover all arcs and nodes?

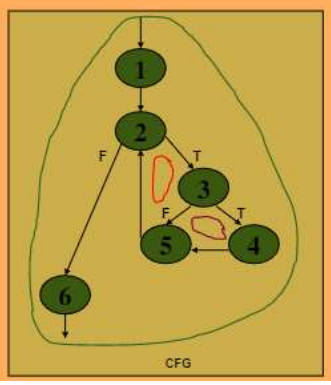


- White Box Testing: Path Based
  - Generalized technique to find out the numb of paths needed (aka cyclomatic complexity) to cover all arcs and nodes in CFG
  - Steps
    1. Draw the CFG for the code frag
    2. Compute the cyclomatic complexity number C for the CFG
    3. Find at most C paths that cover the nodes and arcs in a CFG, aka **Basic Paths Set**
    4. Design test cases to force execution along paths in the **Basic Paths Set**



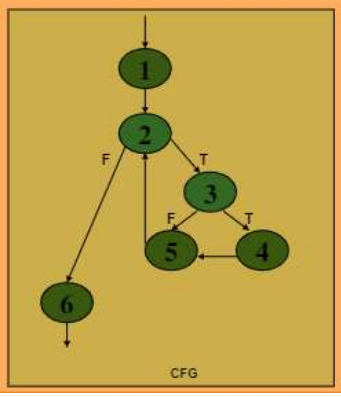


- ## Path Base Testing: Step 2



- **Region**: Enclosed area in the CFG.
  - Do not forget the outermost region.
- In this example:
  - 3 Regions (see the circles with different colors).
  - Cyclomatic Complexity = 3
- Alternative way in next slide.

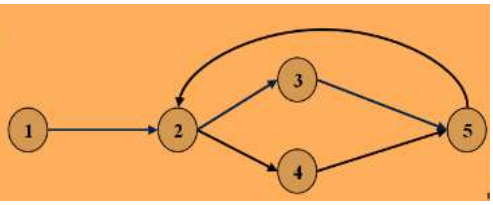
- ## Path Base Testing: Step 2



- **Predicates**:
  - Nodes with multiple exit arcs.
  - Correspond to branch/conditional statement in program.
- In this example:
  - Predicates = 2
    - (Nodes 2 and 3)
  - Cyclomatic Complexity = 2 + 1 = 3

- Pathed Based Testing Step 3
  - Independent Path:
    - An executable/realizable path thru the graph from the starting node to the end node that has not been traversed b4
    - Must move along at least one arc that has not been yet traversed (an unvisited arc)
    - Objective is to cover all statements in a pgrm by independent paths
  - No. of independent paths to discover <= cyclomatic complexity no.
  - Decide the Basis Path set:
    - It's the max set of independent paths in the flow graph
    - Not a unique set

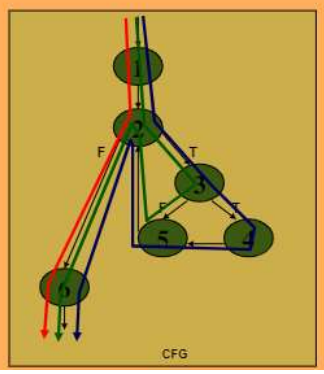
- ## Example



  - Possible first independent path:
    - [1-2-3-5]
    - [1-2-4-5]
    - [1-2-3-5-2-4-5]
  - Only these 3 independent paths, the basis path set is then having 3 paths
  - Alt, if had id'd [1-2-3-5-2-4-5] as first independent path, there would be no more independent paths
  - No. _of independent paths therefore can vary

according to the order id'd them

Path Base Testing: Step 3



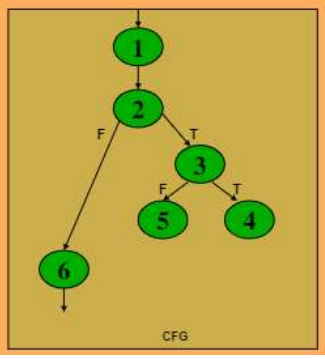
- Cyclomatic complexity = 3.
- Need at most **3** independent paths to cover the CFG.
- In this example:
  - [1 – 2 – 6]
  - [1 – 2 – 3 – 5 – 2 – 6]
  - [1 – 2 – 3 – 4 – 5 – 2 – 6]

- Path Base Testing Step 4
  - Prep a test case for each independent Path
  - In ex:
    - Path [1-2-6]
      - Test case: A = {5, …}, N = 1
      - Expected Output: 5
    - Path: [1-2-3-5-2-6]
      - Test Case: A = {5, 9, …}, N = 2
      - Expected Output: 5
    - Path: [1-2-3-4-5-2-6]
      - Test Case: A = {8, 6, …}, N = 2
      - Expected Output: 6
  - These tests will result a complete decision and statement coverage of the code

Example: Minimum Element

```
min = A[0];                  } 1
I = 1;

while (I < N) {              } 2
     if (A[I] < min)         } 3
        min = A[I];          } 4
     I = I + 1;              } 5
}
print min                    } 6
```



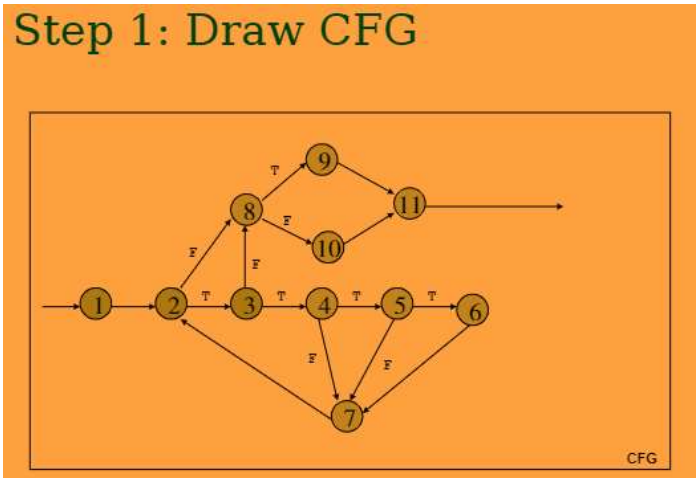
Another Example (how many nodes get a star)

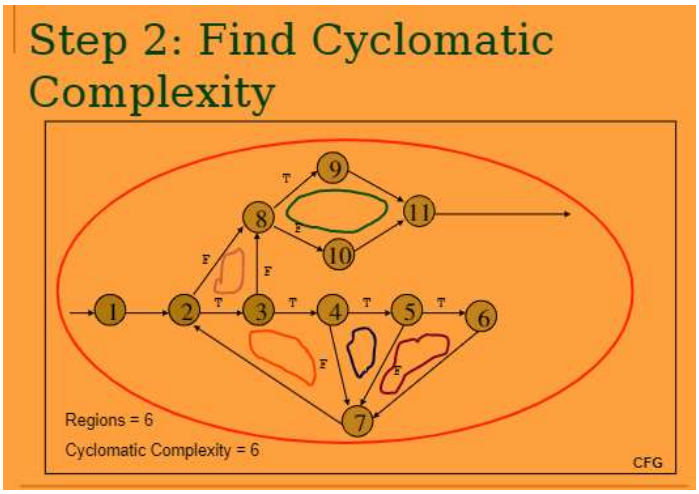
```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {
         if (value[i] >= min && value[i] <= max){
               totalValid += 1; sum += value[i];
      }
         i += 1;
   }
   if (totalValid > 0)
         mean = sum / totalValid;
   else
         mean = -999;
   return mean;
}
```

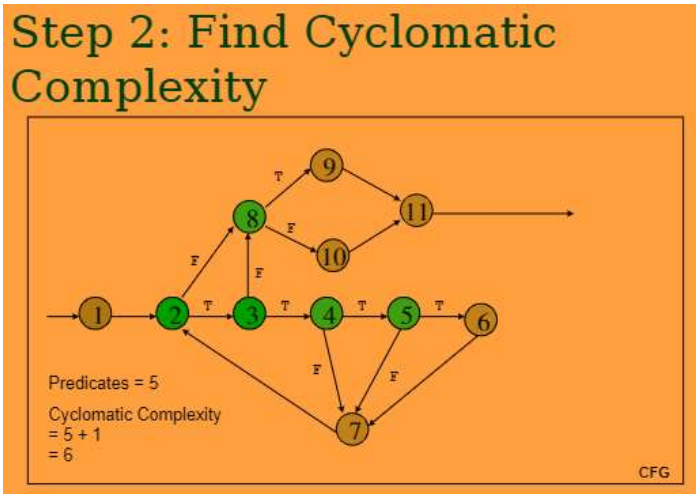
For reference

Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;                    } 1
   while ( i < N && value[i] != -999 ) {
   2          4          3          5
   if (value[i] >= min && value[i] <= max){
   totalValid += 1; sum += value[i];                  6
      }
   i += 1;          7
   }
   if (totalValid > 0)  8
   mean = sum / totalValid;        9
   else
   mean = -999;          10
   return mean;  11
}
```

- 

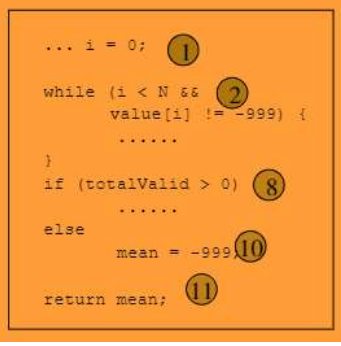
- 

- 

- Step 3: Find Basic Path Set
  - Find max 6 independent paths
  - Usually, simpler paths easier to find a test case
  - But, some of the simpler paths not possible (not realizable)
    - Ex: [1-2-8-9-11]
      - Not realizable (aka impossible)
      - Verify by tracing the code
  - Basic Path Set
    - [1-2-8-10-11]
    - [1-2-3-8-10-11]
    - [1-2-3-4-7-2-8-10-11]
    - [1-2-3-4-5-7-2-8-10-11]
    - [1-(2-6-4-5-6-7)-2-8-9-11]
  - In test case, (...) reps possible repetition

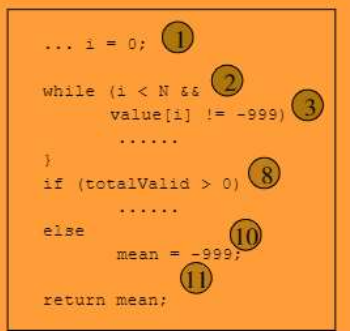
Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 8 – 10 – 11]
- Test Case:
  - `value = {…}` irrelevant.
  - `N = 0`
  - `min, max` irrelevant.
- Expected Output:
  - `average = -999`



Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 8 – 10 – 11]
- Test Case:
  - `value = {-999}`
  - `N = 1`
  - `min, max` irrelevant
- Expected Output:
  - `average = -999`



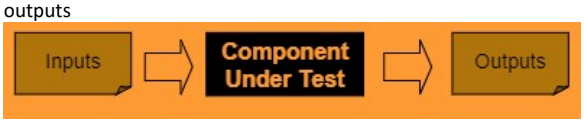
Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is smaller than *min*.
  - `value = { 25 }, N = 1, min = 30, max` irrelevant.
- Expected Output:
  - `average = -999`

- Path:
  - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is larger than *max*.
  - `value = { 99 }, N = 1, max = 90, min` irrelevant.
- Expected Output:
  - `average = -999`

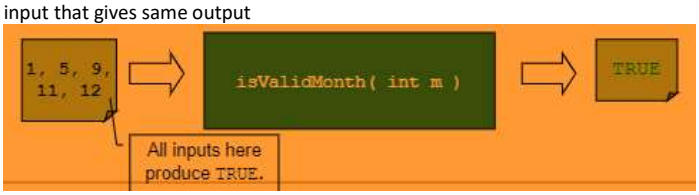
- **Step 4: Derive Test Cases**
  - Path:
    - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
  - Test Case:
    - A single valid value in the `value[ ]` array.
    - `value = { 25 }, N = 1, min = 0, max = 100`
  - Expected Output:
    - `average = 25`

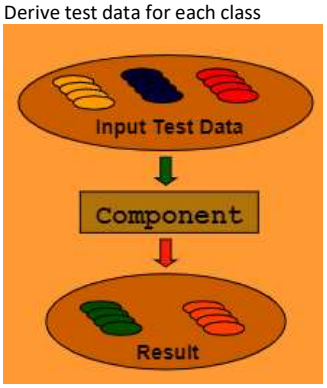
  OR

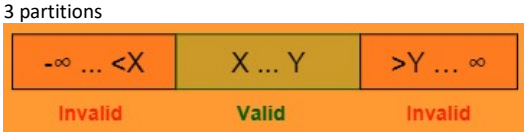
  - Path:
    - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
  - Test Case:
    - Multiple valid values in the `value[ ]` array.
    - `value = { 25, 75 }, N = 2, min = 0, max = 100`
  - Expected Output:
    - `average = 50`
  - Summary: Path Base White Box Testing
    - Simple test that covers all statements and exercises all decisions (conditions)
    - Cyclomatic complexity is an upperbound of the independent paths needed to cover the CFG
      - If more paths are needed, then either cyclomatic complexity is wrong or paths chosen are incorrect
    - Tho picking a complicated path that covers more than one unvisited edge is possible all times, its not encouraged
      - May be hard to design the test case
- Black Box Testing
- Black Box Testing: Intro
  - Test Engineers have no access to the src code/ doc of internal working
  - Black box can be
    - Single unit
    - Subsys
    - Whole sys
  - Tests are based on
    - Specification of the Black Box
    - Providing inputs to the Black box and inspect the outputs
    - 

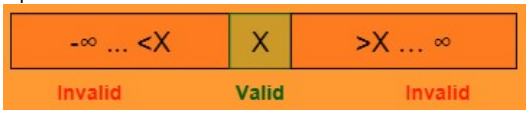
- Test Case Design
  - 2 tech will be covered for black box testing
    - Equivalence partition
    - Boundary value analysis
- Equivalence Partition: Intro
  - To ensure correct behavior of a black box, both valid and invalid cases need to be tested
  - Ex:
    - ```
      boolean isValidMonth(int m)

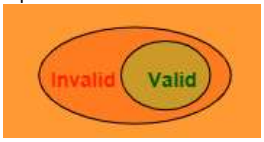
      Functionality: check m is [1..12]
      Output:
            - true if m is 1 to 12
            - false otherwise
      ```
  - Is there better way to test other than testing all int values [-2^{31} , …, 2^{31} -1]?
- Equivalence Partition
  - Experience shows that exhaustive testing is not feasible/ necessary
    - Impractical for most methods
    - An error in code would have caused same failure for many input vals
      - No reason why a val will be treated diff than other
      ```

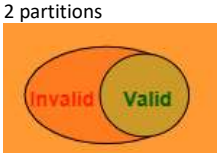
        - Like if val 240 fails testing, then 241 likely to fail as well
    - Better way to choosing test cases
    - Observations
        - For a method, common to have a numb of inputs that gives similar outcomes
        - Testing 1 input should be as good as exhaustively testing all
        - So pick only few test cases from each "category" of input that gives same output
        - 

- Equivalence Partition: Def
    - Partition input data into equivalence classes
    - Data in each equivalence class
        - Likely to be treated equally by a reasonable algorithm
        - Give the same output state, like valid/invalid
    - Derive test data for each class
    - 

- Ex: (isValidMonth)
    - For the isValidMonth ex
        - Input val (1 … 12) should get a similar treatment
        - Input vals less than 1, larger than 12 are 2 other groups
    - 3 partitions
        - [-infinity, …, 0] - invalid
        - [1 … 12] - valid
        - [13 … infinity] - invalid
    - Pick 1 val from each partition as test case
        - Like {-12, 5, 15}
        - Reduce numb of test cases significantly
- Common Partitions
    - If component specifies an input range, [x … y]
        - 3 partitions
        - 

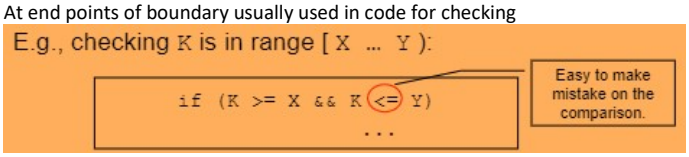
    - If component specifies a single value, [x]
        - 3 partitions
        - 

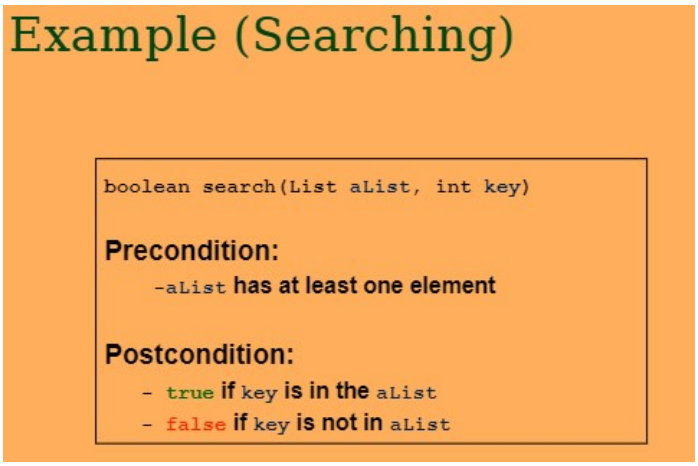
    - If component specifies member(s) of a set:
        - Like traffic light color = {Green, Yellow, Red}, vehicles stop on Red
        - 2 partitions
        - 

- If component specifies a boolean value
  - 2 partitions
  - 

- Combination of Equivalence Classes
  - No. _ of test cases can grow very large when multiple param
  - Ex:
    - Check a phone number w/ following format:
    - (XXX) XXX - XXXX
    - Area Code, Prefix, Suffix
    - Also
      - Area Code Present: Boolean value [True or False]
      - Area Code: 3 digit number [200 … 999]
      - Prefix: 3-digit number not beginning with 0 or 1
      - Suffix: 4-digit number
    - How many equivalence classes are there?
    - Area Code Present: Bool value [True or False]
      - 2 classes:
      - [True]
      - [False]
    - Area Code: 3-digit number [200 … 999] except 911
      - 5 classes:
      - [-infinity … 199]
      - [200 … 910]
      - [911]
      - [912 … 999]
      - [1000 … infinity]
    - Prefix: 3-digit number not beginning with 0 or 1
      - 3 classes:
      - [-infinity … 199]
      - [200 … 999]
      - [1000 … infinity]
    - Suffix: 4-digit number
      - 3 classes:
      - [-infinity … -1]
      - [0000 … 9999]
      - [10000 … infinity]
    - A thorough testing would req us to test all combo of the equivalence classes for each param
    - So tot equivalence classes for the ex should be multiplication of equivalence classes for each input
      - 2 * 5 * 3 * 3 = 90 classes = 90 test cases (!)
    - For critical sys, all combos should be tested to ensure a correct behavior
    - Less stringent testing strat can be used for norm sys
- Reduction of Test Cases
  - Reasonable approach to reduce test cases is
    1. At least one test for each equivalence class for each param, diff equivalence class should be chosen for each param in each test to min the numb of cases
    2. Test all combo (if possible) where a param may affect each other
    3. A few other rando combos
    - Ex: as area code has most classes (like 5), 5 test cases can be defined which simultaneously tries out diff equivalence classes for each param:
Test Case	Area Code Present	Area Code	Prefix	Suffix
1	False	[-∞ …199]	[-∞ … 199]	[-∞ … -1]
2	True	[200 … 910]	[200 … 999]	[0000 … 9999]
3	True	[911]	[1000 … ∞]	[10000 … ∞]
4	True	[912 … 999]	[200 … 999]	[0000 … 9999]
5	True	[1000 … ∞]	[1000 … ∞]	[10000 … ∞]

      E.g., Actual Test Case Data: (True, 934, 222, 4321)
    - In this ex, Area Code Present affects the

interpretation of Area Code, so all combos btwn these 2 params should be tested
    - 2 * 5 = 10 test cases
    - 5 combos already tested in previous test cases, 5 more needed
  - Not counting extra rando combos, this strat reduces numb of test cases to only 10
- Boundary Value Analysis: Intro
  - Found that most errors at boundary of equivalence classes
  - At end points of boundary usually used in code for checking
  - 

  - So, when choosing test data using equivalence classes, boundary vals should be used
  - Complement the equivalence class testing
- Using Boundary Value Analysis
  - If component specifies a range, [x … y]
    - 4 vals to test
      - Valid: X and Y
      - Invalid: Just above X
      - Invalid: Just above Y
    - Like set [1 … 12] could have test data of {0, 1, 12, 13}
  - Similar for open interval (x … y), like x and y not included
    - 4 vals tested
      - Invalid: x and y
      - Valid: just above x
      - Valid: just below y
    - Lise set (100 … 200) could have test data {100, 101, 109, 200}
  - If component specifies a numb of vals
- Fnality Testing
  - Functionality testing of method by black box testing technique
  - Requires method to be well specified
    - Precondition, postcondition, invariant all available
    - Invariant: property that is preserved by the method, like true b4 and after the execution of method
  - For a well specified method / component
    - Use precondition to define equivalence classes, apply boundary value analysis if possible to choose test data from the equivalence classes
    - Use postcondition and the invariant to derive expected results
- 

- Equivalence Classes
  - Sequence w/ a single val
    - Key found
    - Key not found
  - Sequence of multi values
    - Key found
      - First element in sequence

        - Last element in sequence
        - Middle element in sequence
    - Key not found
-

A Test Data for the `search()` method

Test Case	aList	Key	Expected Result
1	[123]	123	True
2	[123]	456	False
3	[1, 6, 3, -4, 5]	1	True
4	[1, 6, 3, -4, 5]	5	True
5	[1, 6, 3, -4, 5]	3	True
6	[1, 6, 3, -4, 5]	123	False

-

Example (Stack – Push Method)

```
void push (Object obj) throws FullStackException
```

Precondition:

```
- ! full()
```

Postcondition:

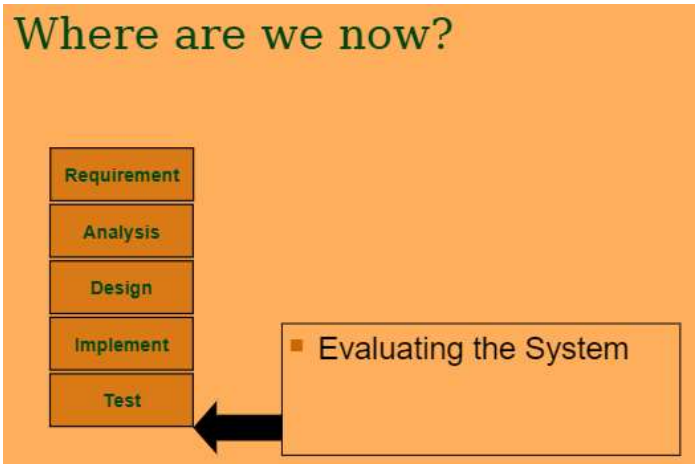
```
- if !full() on entry  then
        top() == obj && size() == old size() + 1
  else throw FullStackException
```

Invariant:

```
- size() >= 0 && size() <= capacity()
```

- Common methods in the `Stack` class:
  - `full()`, `top()`, `size()`, `capacity()`

- Test Data
  - Precondition: Stack isn't full (like bool),
    - 2 equivalence classes can be defined
    - Valid Case: Stack not full
      - Input: a non-full stack, an object obj
      - Expected result:
      -
```
top() == obj
size() == old size() + 1
0 <= size() <= capacity()
```
    - Invalid Case: Stack is full
      - Input: a full stack, an object obj
      - Expected result:
      - `FullStackException` is thrown
```
0 <= size() <= capacity()
```
-

- 
Where are we now?
Requirement
Analysis
Design
Implement
Test
Evaluating the System
- 
Summary
Test Case Design
White Box
Control Flow Graph
Cyclomatic Complexity
Basic Path Testing
Black Box
Equivalence Classes
Boundary Value Analysis
-

L13 Assessment

Thursday, April 13, 2023 4:12 PM

Question 1
10 Points

Which of the following assertions about the cyclomatic complexity is false?

The cyclomatic complexity is used to find out how many independent paths are enough to cover the control flow graph;
The cyclomatic complexity equals to the number of regions in the control flow graph;
The cyclomatic complexity equals to the number of predicates plus 1;
The cyclomatic complexity measures the time complexity of the given program;
The number of independent paths to discover is always less than or equal to the cyclomatic complexity number.

Question 2
10 Points

Consider the following program fragments. For each program fragment, draw a flow graph, identify the independent paths, and, for each path, give examples of inputs that would cause the path to be executed.

(a)

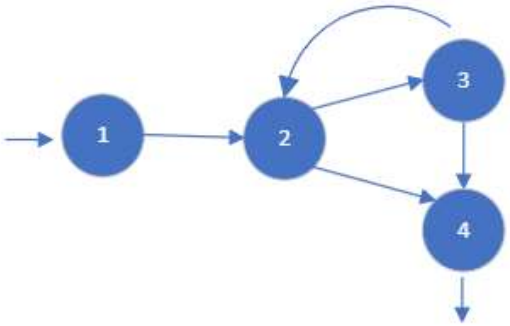
```
z= readNumber();
y =readNumber();
x = 1;
j = 1;
while (y>= j) {
j = j + 1;
x = x * z;
}
writeNumber(x);
```

```
z= readNumber();
y =readNumber();
x = 1;
j = 1;

while (y>= j) {
      j = j + 1;
      x = x * z;
}
writeNumber(x);
```

```
z= readNumber();   1
y =readNumber();   1
x = 1;             1
j = 1;             1

while (y>= j) {    2
      j = j + 1;   3
      x = x * z;   3
}
writeNumber(x);    4
```



Regions = 3

Predicates = 2

2 + 1 = 3

So Cyclocomplexity = 3

Independent paths are:

A) [1-2-4]

Test cases: y = {0, -1, …}

z = {...}

B) [1-2-3-4]

Test cases: y = {1}

z = {...}

C) [1-2-3-2-4]

Test cases: y = {10, ...}

z = {...}

(b)

x = readNumber();

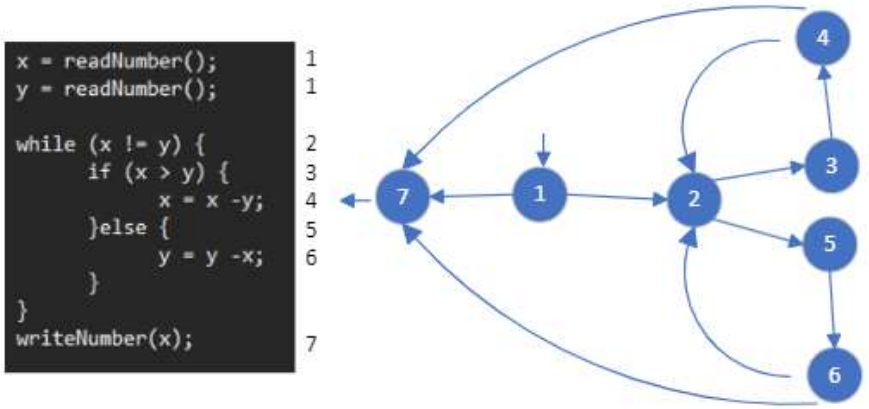
y = readNumber();

while (x != y) {

```
if (x > y) {
x = x -y; }
else
{ y = y -x;}
}
writeNumber(x);
```

```
x = readNumber();
y = readNumber();

while (x != y) {
      if (x > y) {
            x = x -y;
      }else {
            y = y -x;
      }
}
writeNumber(x);
```



Regions = 5
Predicates = 4
4 + 1 = 5
So Cyclocomplexity = 5

Independent Paths are:
A) [1-7]
Test cases: x = {...} = y
B) [1-2-3-4-7]
Test cases: x = {6, 12, ...}
y = {3, 6, ...}
Where y is half of x
C) [1-2-5-6-7]
Test cases: x = {3, 6, ...}
y = {6, 12, ...}
Where y is double of x
D) [1-2-3-4-2-3-4-7]
Test cases: x = {9, 18, 81, …}
y = {3, 6, ...}

Where y is a factor of x

E) [1-2-5-6-2-5-6-7]

Test cases: x = {3, 6, ...}

y = {9, 18, 81, ...}

Where y is a multiple of x

Question 3
10 Points

Read the following Java code fragment for a method which attempts to calculate the number of students having the mark between 1 and 4.

ns = 0;

I = 0;

while (I <= N) {

if (mark[I] >= 1.0 && mark[I] < 4.0)

ns++;

I++;

}

print ns;

```
ns = 0;
I = 0;

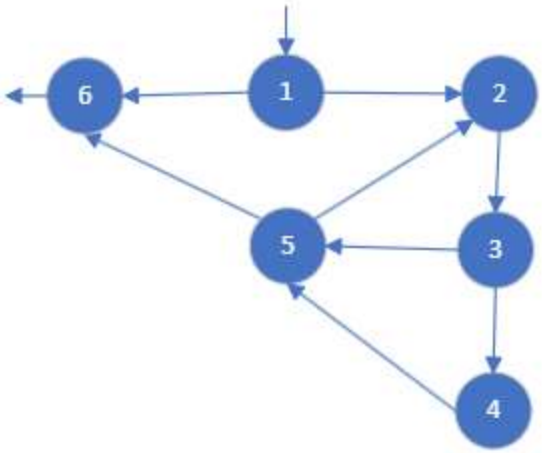
while (I <= N) {
      if (mark[I] >= 1.0 && mark[I] < 4.0)
            ns++;
      I++;
}

print ns;
```

```
ns = 0;                                          1
I = 0;                                           1

while (I <= N) {                                 2
      if (mark[I] >= 1.0 && mark[I] < 4.0)       3
            ns++;                                4
      I++;                                       5
}

print ns;                                        6
```



Regions = 4

Predicates = 3

3 + 1 = 4

So Cyclocomplexity = 4

Independent Paths are:

A) [1-6]

Test cases: mark = {...}

N = {-1, -2, …}

B) [1-2-3-5-6]

Test cases: mark = {0, 4, 5, 6, …}

N = 0

C) [1-2-3-5-2-3-5-6]

Test cases: mark = {0, 4, 5, 6, …}

N = {1, 2, …}

D) [1-2-3-4-5-6]

Test cases: mark = {1, 2, 3}

N = 0

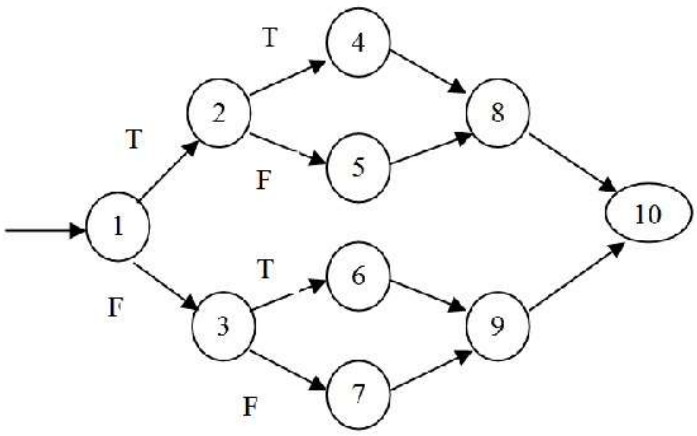
(i) Draw a control flow graph.

(ii) Based on the control flow graph, find the cyclomatic complexity and list all the (linearly) independent paths.

(iii) Prepare a test case for each independent path.

Question 4
10 Points

Choose the right answer. The cyclomatic complexity of the below graph is:



4
3
2
5