Multiple-Choice Questions:

1. True or false? Generally, in practice, developers exhaustively test software.
    a. True
    b. False

2. True or false? All "real" software contains bugs.
    a. True
    b. False

3. Which of the following is not a desirable quality of a unit test?
    a. No I/O
    b. Fast
    c. Non-deterministic
    d. Tests one property
    e. None of the above

4. Which of the following is true of *exhaustive testing*?
    a. Generally infeasible in practice
    b. Tests all possible inputs
    c. Typically results in an intractably large set of test cases even for small programs
    d. All of the above
    e. None of the above

5. Which of the following is not a difference between unit tests and integration tests?

    a. Unit tests should not perform I/O, whereas integration tests may do so

    b. Unit tests should be deterministic, whereas integration tests may have non-determinism

    c. Unit tests should be fast (less than half a second), whereas integration tests may be slower

    d. Unit tests must be black-box tests, whereas integration tests must be white-box tests

    e. None of the above (they are all differences)

6. Which of the following is not a difference between black-box and white-box testing?

    a. Black-box tests are based only on the interface of a component, whereas white-box tests are based on the implementation

    b. Black-box tests often focus on boundary cases, whereas white-box tests tend not to

    c. White-box tests often aim to achieve particular levels of code-coverage, whereas black-box tests do not

    d. White-box tests are made by programmers, whereas black-box tests are made by ordinary users

    e. None of the above (they are all differences)

7. In _____, you hook everything together and treat the system like a black box.

    a. test-driven development

    b. system testing

    c. unit testing

    d. integration testing

    e. None of the above

Solutions:

1. b
2. a
3. c
4. d
5. d
6. d
7. b

Problem:

Consider these code fragments.

a. `end`

b. `get :index`

c. `assert_redirected_to car_path(assigns(:car))`

d. `assert_template :index`

e. `assert_template :new`

f. `assert_not_nil assigns(:cars)`

g. `get :new`

h. `test "should get index" do`

i. `post :create, car:{make:@car.make, model:@car.model, year: @car.year}`

j. `assert_response :success`

Using the above fragments, create a functional test for the "index" page of a car-themed web app. The test should make sure (1) that the HTTP response does not report an error, (2) that the correct ERB is rendered (index.html.erb), and (3) that the call to `Car.all` in the controller, which sets the `@cars` instance variable, does not fail and return nil. Note that your answer should use only 6 of the above fragments.

_____

_____

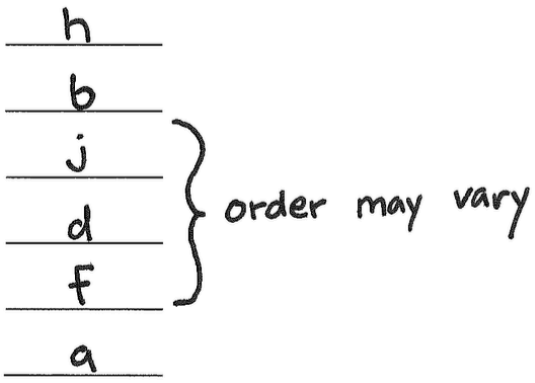
_____

_____

_____

_____

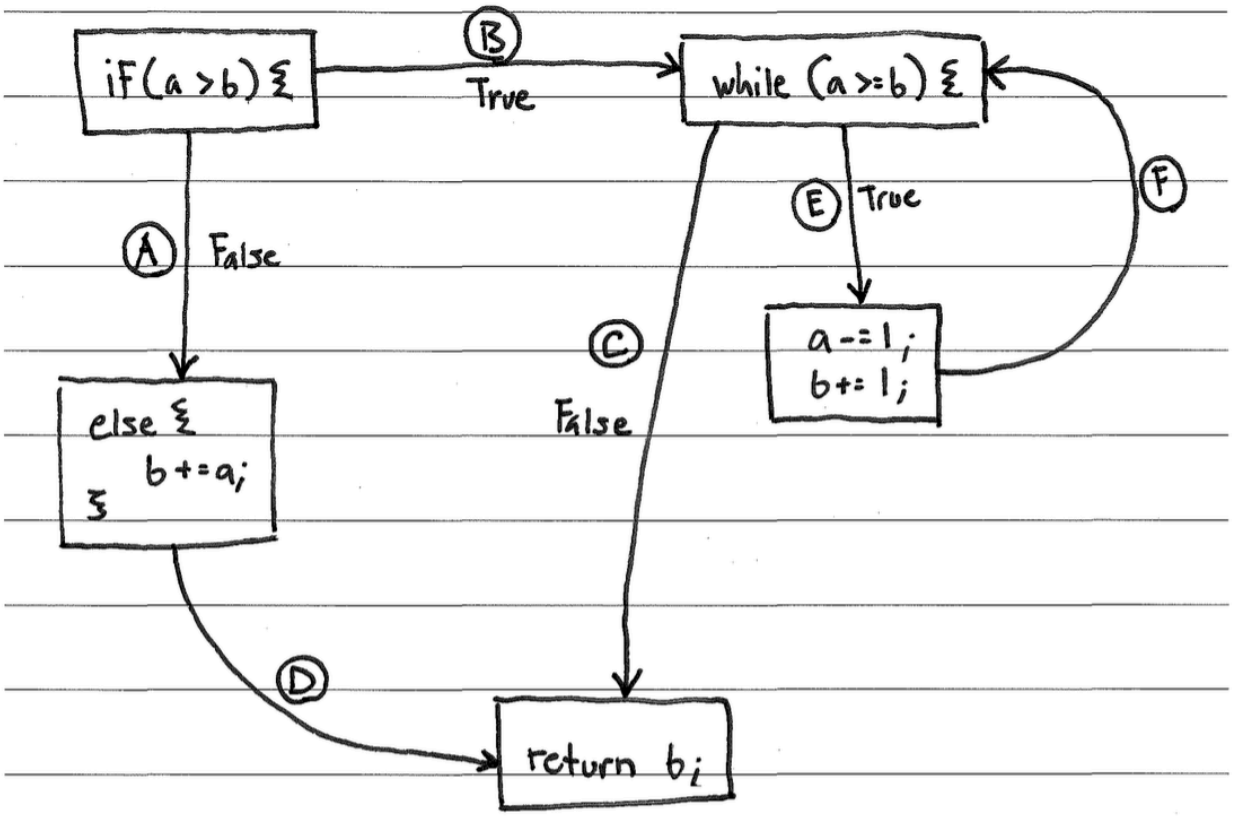
Solution:



Problem: Draw a control flow diagram for this function. Label each edge with an uppercase letter.

```
int funWithNumbers(int a, int b) {
    if (a > b) {
        while (a >= b) {
            a -= 1;
            b += 1;
        }
    } else {
        b += a;
    }
    return b;
}
```

Solution:


iF(a >b) {
B
True
while (a >=b) {
A
False
E
True
F
a -=1;
b+= 1;
C
False
else {
b+=a;
}
D
return b;

Problem: Fill in the table below with a test suite that provides *path coverage* of the code from the previous question. Cover no more than 2 iterations of the loop. In the covers column, list the relevant labeled items in your CFG that each test case covers. If there is some part of the coverage that is impossible to cover, then list it in the covers column, and put "N/A" in the associated x and y cells. Some cells in the table may be left blank.

Input		**Covers**
x	**y**	

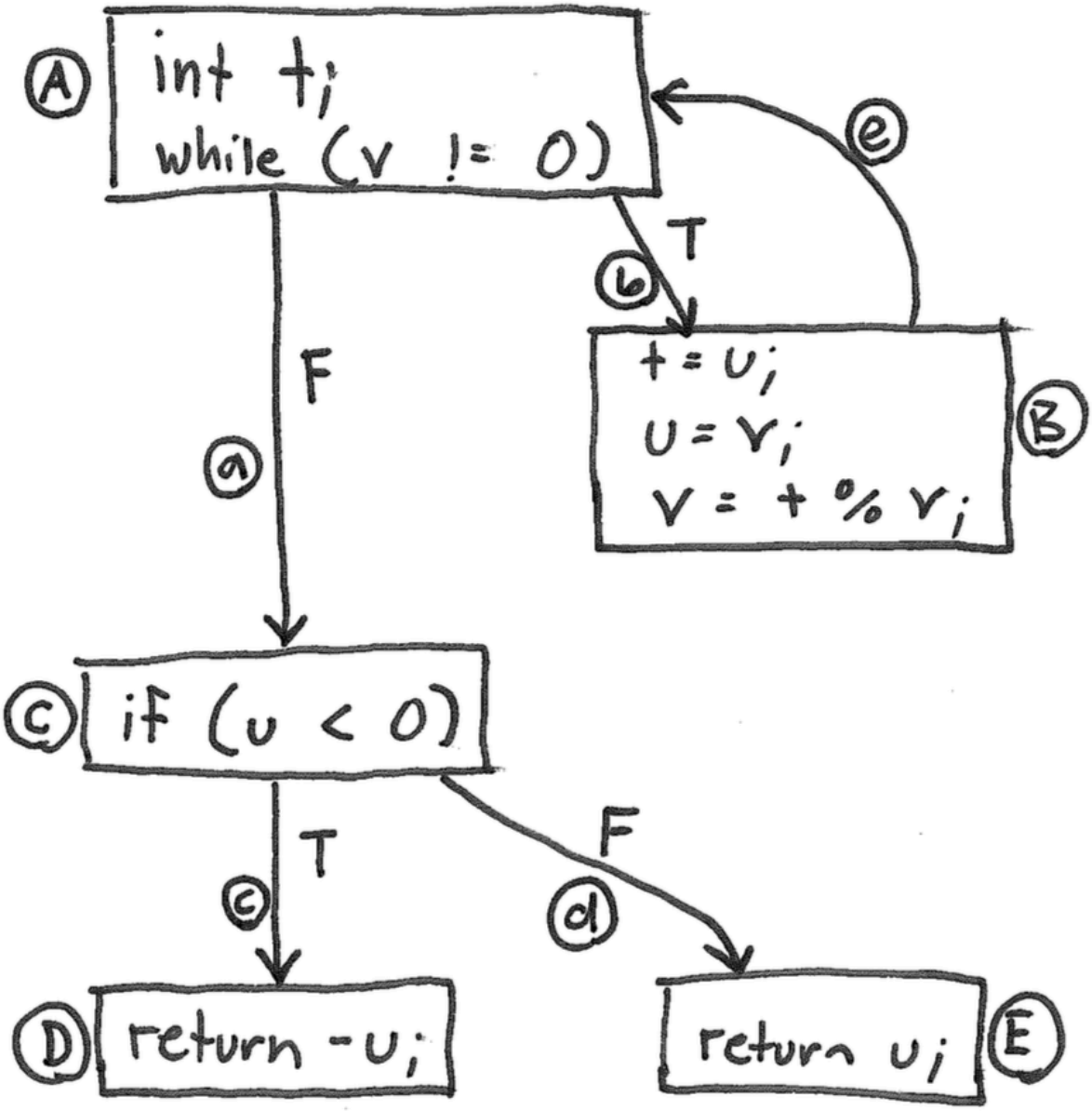
Solution:

Input		Covers
x	y	
1	2	AD
N/A	N/A	BC
1	0	BEFC
4	2	BEFEFC

Problem: Draw a control flow diagram for this function. Label each node in the graph with a capital letter, and label each edge with a lowercase letter.

```
int blammo(int u, int v) {
  int t;
  while (v != 0) {
    t = u;
    u = v;
    v = t % v; // Recall that % computes remainder of t/v
  }
  if (u < 0) { return -u; }
  return u;
}
```

Solution:


A
int t;
while (v != 0)
e
T
b
t = u;
u = v;
v = t % v;
B
F
a
c
if (u < 0)
T
F
c
d
D
return -u;
return u;
E

Problems:

2. Fill in the table below with a test suite that provides statement coverage of the "blammo" code. In the covers column, list the relevant labeled items in your CFG that each test case covers. Some cells in the table may be left blank.

Input		Covers
u	v	

3. Fill in the table below with a test suite that provides path coverage of the "blammo" code. Cover no more than 1 iteration of the loop. In the covers column, list the relevant labeled items in your CFG that each test case covers. Some cells in the table may be left blank.

Input		Covers
u	v	

Solutions:

1.

Input		Covers
u	v	
2	2	A, B, C, E
-1	0	A, C, D

2.

Input		Covers
u	v	
-1	0	a, c
0	0	a, d
-2	-2	b, e, a, c
2	2	b, e, a, d

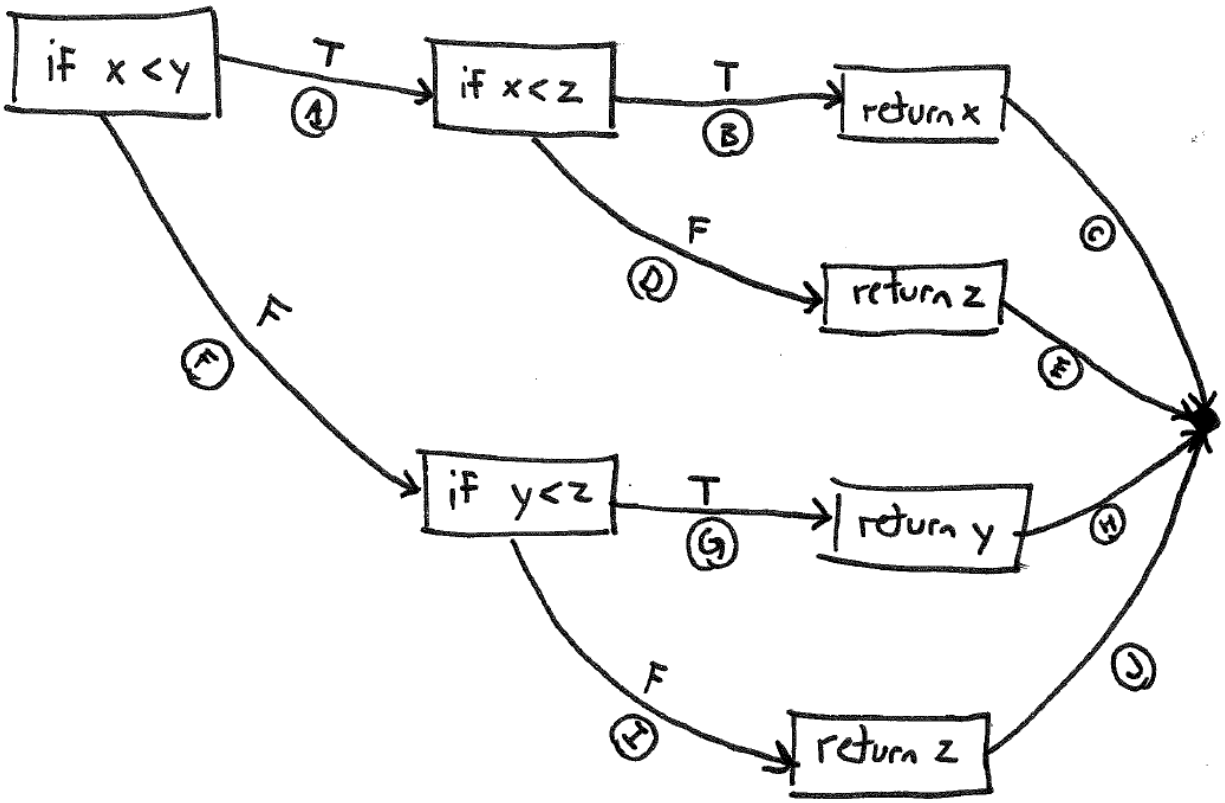
Paths:

a, c
a, d
b, e, a, c
b, e, a, d

Problem: Draw a control-flow graph for the following function. Label each edge in the graph with an uppercase letter.

```
def min_of_three(x, y, z)
  if x < y then
    if x < z then
      return x
    else
      return z
    end
  else
    if y < z then
      return y
    else
      return z
    end
  end
end
```

Solution:


if x<y
T
1
if x<z
T
B
return x
C
F
D
return z
E
F
if y<z
T
G
return y
H
F
I
return z
J

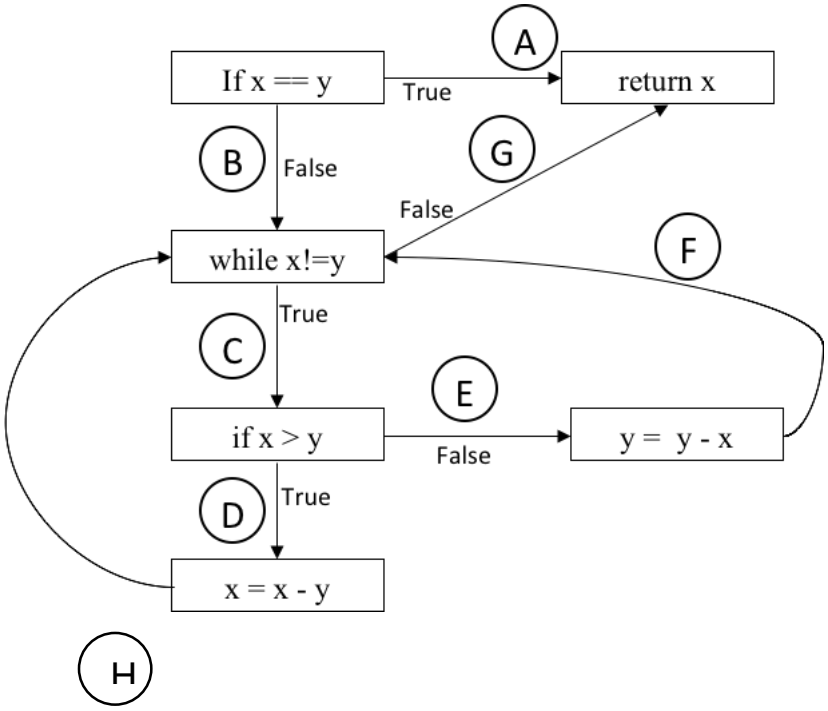
Problem: Fill in the table below with a test suite that provides path coverage of the `min_of_three` function from the previous question. In the covers column, list the relevant labeled edges in your CFG that each test case covers. Some cells in the table may be left blank.

Input			Expected Output	Covers
x	y	z		

Solution:

Input			Expected Output	Covers
x	y	z		
1	2	2	1	A, B, C
2	3	1	1	A, D, E
2	1	2	1	F, G, H
3	2	1	1	F, I, J

Consider the following control-flow graph for a `gcd` function in answering the questions below.



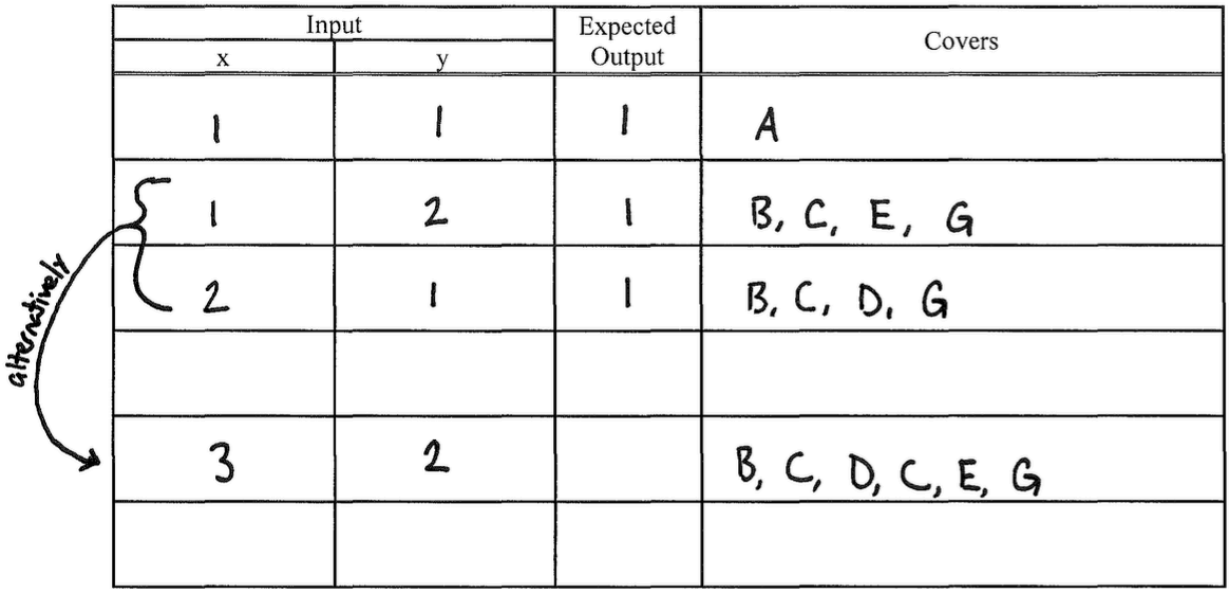
Problem: Fill in the table below with a test suite that provides condition coverage of the `gcd` function (see control-flow graph above). In the Covers column, list the relevant labeled edges in the CFG that each test case covers. Some cells in the table may be left blank.

Input		Expected Output	Covers
x	y		

Problem: Fill in the table below with a test suite that provides path coverage of the `gcd` function (see control-flow graph above). In the Covers column, list the relevant labeled edges in the CFG that each test case covers. Some cells in the table may be left blank. You need only cover executions that involve 1 iteration of the loop.

Input		Expected Output	Covers
x	y		

Solution: Condition Coverage



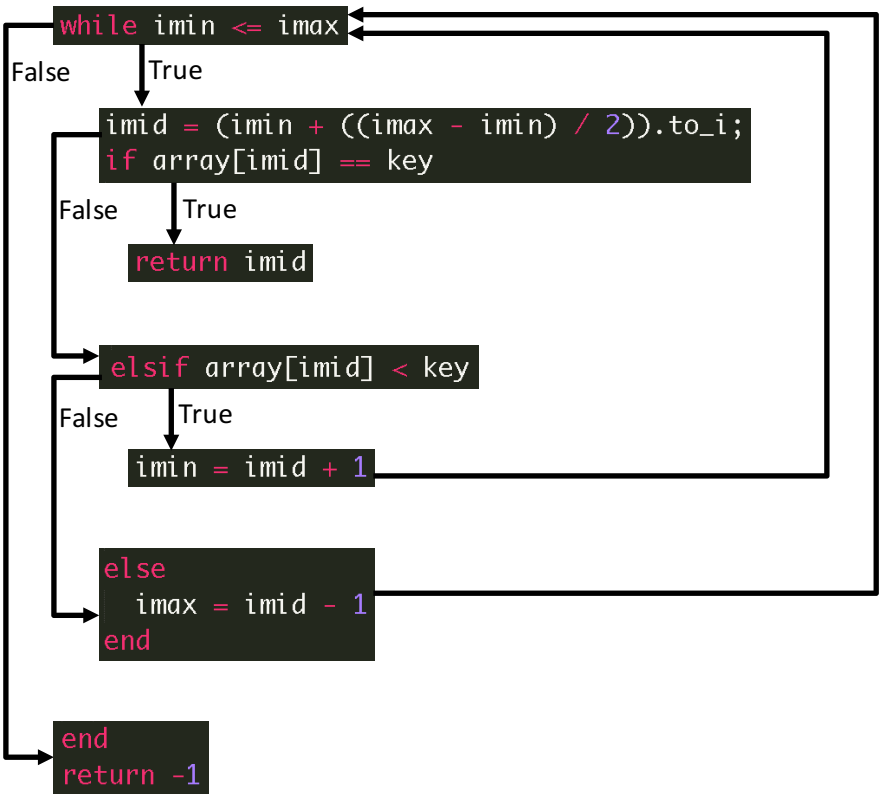
Input		Expected Output	Covers
x	y		
1	1	1	A
1	2	1	B, C, E, G
2	1	1	B, C, D, G
3	2		B, C, D, C, E, G

Solution: Path Coverage

Input		Expected Output	Covers
x	y		
1	1	1	A
2	1	1	B, C, D, H, G
1	2	1	B, C, E, F, G
			B, G ← not possible

Consider this binary-search function and its associated control-flow graph.

```
def binary_search(array, key, imin, imax)
  while imin <= imax
      imid = (imin + ((imax - imin) / 2)).to_i;
      if array[imid] == key
        return imid
      elsif array[imid] < key
        imin = imid + 1
      else
        imax = imid - 1
      end
  end
  return -1
end
```



Problems:

Consider the following test cases for the `binary_search` function.

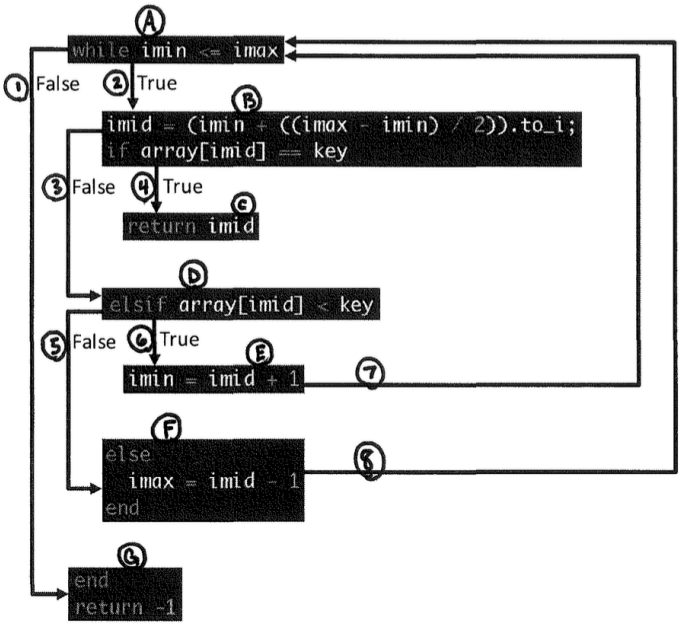
	array	key	imin	imax
a.	`[1]`	0	0	0
b.	`[1]`	1	0	0
c.	`[1]`	2	0	0
d.	`[1, 2, 3]`	1	0	2
e.	`[1, 2, 3]`	2	0	2
f.	`[1, 2, 3]`	3	0	2
g.	`[1, 2, 3]`	1	2	0
h.	`[1, 2, 3]`	2	2	0
i.	`[1, 2, 3]`	3	2	0

1. Select tests from the above to create a test suite that provides statement coverage of the `binary_search` function. Your suite should contain the minimum number of tests to provide the coverage.

2. Select tests from the above to create a test suite that provides condition coverage of the `binary_search` function. Your suite should contain the minimum number of tests to provide the coverage.

3. Select tests from the above to create a test suite that provides path coverage of the `binary_search` function. Cover only paths that contain one loop iteration or fewer (i.e., no path should enter the loop more than once). Your suite should contain the minimum number of tests to provide the coverage.

Solutions:



	array	key	imin	imax	Statements Covered	Edges Covered
a.	[1]	0	0	0	ABDFAG	23581
b.	[1]	1	0	0	ABC	24
c.	[1]	2	0	0	ABDEAG	23671
d.	[1, 2, 3]	1	0	2	ABDFABC	235824
e.	[1, 2, 3]	2	0	2	ABC	24
f.	[1, 2, 3]	3	0	2	ABDEABC	236724
g.	[1, 2, 3]	1	2	0	AG	1
h.	[1, 2, 3]	2	2	0	AG	1
i.	[1, 2, 3]	3	2	0	AG	1

1. a, F or c, d
(Need to cover statements A, B, C, D, E, F, G)

2. a, F or c, d
(Need to cover edges 1, 2, 3, 4, 5, 6)

3. (g|h|i), (b|e), c, a

Any 1 of these — Any 1 of these

Possible paths	Tests that cover
1	g, h, i
24	b, e
23671	c
23581	a