Exercise 1

**ESTIMATED TIME TO COMPLETE: 4 minutes**

Consider the following code specification:

def size(aSet):

"""

aSet is a collection of objects, which might be empty.

Objects are assumed to be of the same type.

"""

Here is a set of possible test cases to include in a black box test suite. Indicate which of the following conditions would make a good black box test suite for the function size by clicking on the appropriate choice(s).

[Review: Black Box Test Suites](https://courses.edx.org/xblock/block-v1:MITx+6.00.1x+1T2021+type@vertical+block@9477f816676c482eb8ad4d00b695ebb0?show_title=0&show_bookmark_button=0&recheck_access=1&view=student_view&format=Finger%20Exercises)

Empty set

Set of size 1

Set of odd size

Set of even size

Set of size greater than 1

Set whose size is a prime number

**Explanation:**

A good black box test suite would contain tests for the following conditions: Empty set, Set of size 1, and Set of size greater than 1.

Black-box testing is a method of software testing that tests the functionality of an application. Recall from the lecture that a way to think about black-box testing is to look at both the paths through the specification and the possible boundary cases. In this example, the boundary cases all have to do with the size of aSet. Specifically these boundary cases are when aSet contains zero, one, or many items.

The remaining conditions would not further test the functionality of the size function because an odd, even, or prime sized set are all sets of size greater than 1. Nothing in the function specification suggests there is anything special or unique about odd, even, or prime sized sets, so testing those cases specifically simply repeats the test "Set of size greater than 1".

Exercise 2

**ESTIMATED TIME TO COMPLETE: 5 minutes**

Consider the following code specification:

def union(set1, set2):

"""

set1 and set2 are collections of objects, each of which might be empty.

Each set has no duplicates within itself, but there may be objects that

are in both sets. Objects are assumed to be of the same type.

This function returns one set containing all elements from

both input sets, but with no duplicates.

"""

Indicate which of the conditions below would combine to make a good black box test suite for the function union by selecting the appropriate choice(s).

set1 is an empty set; set2 is an empty set

set1 is an empty set; set2 is of size greater than or equal to 1

set1 is of size greater than or equal to 1; set2 is an empty set

set1 and set2 are both nonempty sets which do not contain any objects in common

set1 and set2 are both nonempty sets which contain objects in common

**Explanation:**

A good black box test suite would contain tests for all of the given conditions! Black-box testing tests the functionality of an application, by looking at the paths through its specifications.

According to the specifications, the possibilities for set1 and set2 are as follows: both sets are empty; one of the sets is empty and one has at least one object; both sets are not empty. The tests list all the combinations of those possibilities for set1 and set2.

### Exercise 3

**ESTIMATED TIME TO COMPLETE: 3 minutes**

Consider the following function definition:

def maxOfThree(a,b,c) :

"""

a, b, and c are numbers

returns: the maximum of a, b, and c

"""

if a > b:

bigger = a

else:

bigger = b

if c > bigger:

bigger = c

return bigger

Assume that maxOfThree is called with numbers as arguments.

Which of the following test suites would make a path-complete glass box test suite for maxOfThree?

[Review: Glass Box Test Suites](https://courses.edx.org/xblock/block-v1:MITx+6.00.1x+1T2021+type@vertical+block@64a310219b6648998b884e99d0588f61?show_title=0&show_bookmark_button=0&recheck_access=1&view=student_view&format=Finger%20Exercises)

Test Suite A: maxOfThree(2, -10, 100), maxOfThree(7, 9, 10), maxOfThree(6, 1, 5), maxOfThree(0, 40, 20)

Test Suite B: maxOfThree(10, 100, -20), maxOfThree(99, 0, 20), maxOfThree(1, 60, 300)

Test Suite C: maxOfThree(0, 0, 0), maxOfThree(-3, -10, -1), maxOfThree(10, 30, 100), maxOfThree(0, -9, 11), maxOfThree(-10, 0, 30)

**Explanation:**

Recall from the lecture that a path-complete glass box test suite would find test cases that go through every possible path in the code. In this case, that means finding all possibilities for the conditional tests a > b and c > bigger. So, we end up with four possible paths that correspond to Test Suite A:

* Case 1: a > b and c > bigger - this corresponds to test maxOfThree(2, -10, 100).
* Case 2: a > b and c <= bigger - this corresponds to test maxOfThree(6, 1, 5)
* Case 3: a <= b and c > bigger - this corresponds to test maxOfThree(7, 9, 10).
* Case 4: a <= b and c <= bigger - this corresponds to test maxOfThree(0, 40, 20)

Test Suite B seems to explicitly test each of a, b, and c being the max of the three numbers, but this does not go through all possible paths in the code.

Test Suite C seems to be a good sampling of the space of input numbers, but it does not go through all possible paths in the code.

### Exercise 4

**ESTIMATED TIME TO COMPLETE: 4 minutes**

Consider the following function definition:

def union(set1, set2):

"""

set1 and set2 are collections of objects, each of which might be empty.

Each set has no duplicates within itself, but there may be objects that

are in both sets. Objects are assumed to be of the same type.

This function returns one set containing all elements from

both input sets, but with no duplicates.

"""

if len(set1) == 0:

return set2

elif set1[0] in set2:

return union(set1[1:], set2)

else:

return set1[0] + union(set1[1:], set2)

Assume that union is called with strings as arguments.

Please select the best glass box test suite for the function union from the following options:

Test Suite A: union('',''), union('','a'), union('','ab'), union('a',''), union('a','b'), union('c','ab'), union('de',''), union('ab','c'), union('cd','ab')

Test Suite B: union('abc',''), union('abc','a'), union('abc','ab'), union('abc','d'), union('abc', 'abcd')

Test Suite C: union('','abc'), union('a','abc'), union('ab','abc'), union('abc','abc')

Test Suite D: union('','abc'), union('a','abc'), union('ab','abc'), union('d','abc')

**Explanation:**

A good glass box test suite would try to test a good sample of all the possible paths through the code. So, it should contain tests that test when set1 is empty, when set1[0] is in set2, and when set1[0] is not in set2. The test suite should also test when the recursion depth is 0, 1, and greater than 1.

Remember that glass box testing is a method of software testing that tests the internal structures and workings of a piece of code. When we look at the code for union, we see a set of conditionals that ask about set1. Thus a good glass box test suite will contain tests that match the following lines from the conditional statements in the code:

* len(set1) == 0 - matched by the test union('', 'abc')
* set1[0] in set2 - matched by the test union('a', 'abc')
* set1[0] not in set2 (this is the else: of the conditional) - matched by the test union('d', 'abc')

In addition, because union is a recursive function, we should make sure our test set considers a recursion depth of 0, 1, and many. In this case, recursion depth is covered by some of the tests we've already chosen:

* Recursion depth = 0 - matched by the test union('', 'abc')
* Recursion depth = 1 - matched by the tests union('a', 'abc'), union('d', 'abc')
* Recursion depth > 1 - matched by the test union('ab', 'abc')

Note that this test suite is NOT path complete because it would take essentially infinite time to test all possible recursive depths.

Let's examine now why the other test suites weren't as good as Test Suite D:

* Test Suite A looks at a good sampling of set sizes for set1 and set2, but does not explore the if-else paths in the code. set1 never contains any element in set2.
* Test Suite B does not explore the paths in the code because it never varies the contents of set1.
* Test Suite C does not contain a test that explores the path when set1 has an element that is not in set2.

### Exercise 5

**ESTIMATED TIME TO COMPLETE: 3 minutes**

Consider the following function definition:

def foo(x, a):

"""

x: a positive integer argument

a: a positive integer argument

returns an integer

"""

count = 0

while x >= a:

count += 1

x = x - a

return count

Please select the best glass box test suite for the function foo from the following options.

Test Suite A: foo(2, 5), foo(5, 6), foo(9, 7)

Test Suite B: foo(10, 3), foo(1, 4), foo(10, 6)

Test Suite C: foo(100, 5), foo(96, 5), foo(22, 5)

correct

**Explanation:**

In glass box testing, we try to sample as many paths through the code as we can. In the case of loops, we want to sample three general cases:

1. Not executing the loop at all.
2. Executing the loop exactly once.
3. Executing the loop multiple times.

Test Suite B has cases that explores all three loop cases.

Test Suite A only explores paths that execute the loop 0 or 1 times.

Test Suite C only explores paths that execute the loop more than 1 time.

Exercise 6

**ESTIMATED TIME TO COMPLETE: 3 minutes**

Consider the following function definition:

def rem(x, a):

"""

x: a non-negative integer argument

a: a positive integer argument

returns: integer, the remainder when x is divided by a.

"""

if x == a:

return 0

elif x < a:

return x

else:

rem(x-a, a)

When we call

rem(2, 5)

the shell returns 2. When we call

rem(5, 5)

the shell returns 0. But when we call

rem(7, 5)

the shell does not return anything! Using this information, choose what line of code should be changed from the following choices:

if x == a:

return 0

elif x < a:

return x

else:

rem(x-a, a)

How should this line be rewritten?

  correct

return rem(x-a, a)

**Explanation:**

In the conditional, the final clause contains a recursive call that we never return the results of! Thus the final line - rem(x-a, a) - simply needs to be modified such that we return its value. Easy enough! The solution is thus

return rem(x-a, a)

Exercise 7

2/2 points (graded)

**ESTIMATED TIME TO COMPLETE: 3 minutes**

Consider the following function definition:

def f(n):

"""

n: integer, n >= 0.

"""

if n == 0:

return n

else:

return n \* f(n-1)

When we call f(3) we expect the result 6, but we get 0.

When we call f(1) we expect the result 1, but we get 0.

When we call f(0) we expect the result 1, but we get 0.

Using this information, choose what line of code should be changed from the following choices:

if n == 0:

return n

else:

return n \* f(n-1)

correct

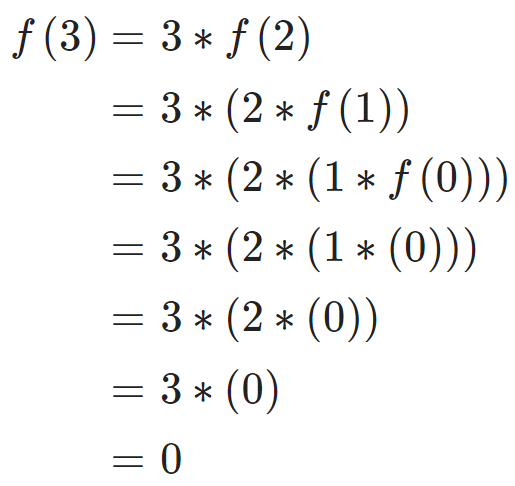
How should this line be rewritten?



return 1

**Explanation:**

This is a function known as [factorial](http://en.wikipedia.org/wiki/Factorial) - the product of all the numbers from 1 through n. The base case of factorial is 0! = 1, but the original code was written with the base case 0! = 0. You can see why the original code was broken by writing out the recursive expansion of f(3):



The fixed version of the code puts the line return 1, instead of return n, when n == 0. We can see that this modified version of the code fixes the factorial function by again writing out the recursive expansion of f(3):

