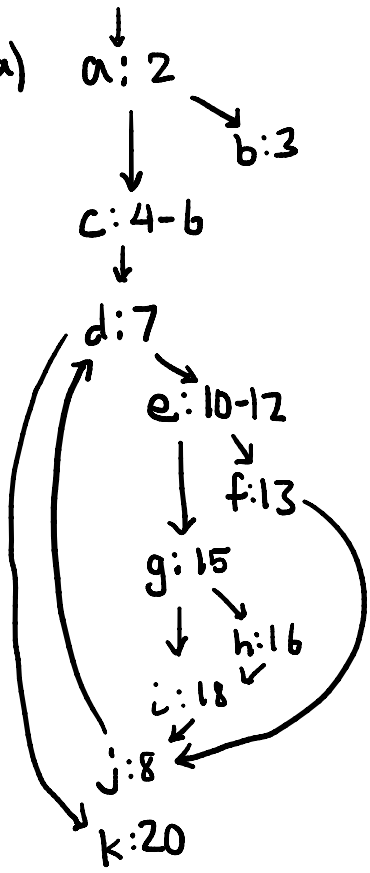
SE465 Final W2015

1.
a) [Not covered]
b) Threading errors, program logic errors (program functionality)
c) Uninitialized memory usage
d) [idk come up with something]
e) tkfuzz, libFuzzer
f) [Not covered] [SE350 moment]

2.
a) Yes. The set of all simple test paths starting at N_0 and ending at N_f will cover all nodes, since all nodes will either be in N_0, N_f, or some path between the two.
b) Yes. PPC covers all prime paths, which are just simple paths of maximal length.

3. a)



b) Pretend this is a CFG

def(a) = { pancakes }
use(a) = { pancakes }

def(b) = use(b) = def(f) = use(f) = def(k) = use(k) = {}

def(c) = { i, the_spot }
use(c) = { pancakes }

def(d) = {}
use(d) = { i }

def(e) = { the_spot }
use(e) = { pancakes, i, the_spot }

def(g) = {}
use(g) = { the_spot }

def(h) = { pancakes } (use is before def)
use(h) = { pancakes, the_spot }

def(i) = { pancakes } (use is before def)
use(i) = { pancakes, i }

def(j) = { i }
use(j) = { i }

c) pancakes is defined in nodes a, h, i
pancakes is used in nodes a, c, e, h, i
<u>du paths</u>
ac, acde, acdegh, acdegi, hi, ijde, ijdegh, ijdegi

AUC is the set of all du paths, since we need one path from every du pair and there is only one path for every du pair.

d) Test paths: acdeghijdegijdk, acdegijdeghijdk

4.

a) Total statement coverage: TC6, TC1, TC2, TC5, TC3, TC4
Additional statement coverage: TC6, TC2, TC4, TC1, TC3, TC5

b) For this program, additional statement coverage is better. We achieve total statement coverage after the first 3 tests with this scheme, whereas in total statement coverage we don't achieve total statement coverage until all 6 test cases have run.

5.

a)

Predicate #	Line Number	Predicate
1	6	fin == 0 || midterm == 0
2	10	fin > 90
3	12	weightedMF < 50

b)

Predicate #	Reachability Condition
1	true
2	fin != 0 && midterm != 0
3	fin <= 90 && fin != 0 && midterm != 0
4	fin <= 90 && weightedMF >= 50 && fin != 0 && midterm != 0

c)

Predicate #	P	A	M	F	P	A	M	F
1	0	0	0	0	0	0	100	100
2	0	0	100	100	0	0	40	40
3	0	0	40	40	0	0	80	80

6.

a) a=[-1, 1], aLength=2
After 1 iteration of the for loop, we have that max=0. However, this is an incorrect internal state, since our current max should be -1. However, 1 > 0, so the mutant returns the correct value of 1.

b) a=[-1], aLength=1
The mutant returns 0, which is incorrect: the expected output is -1.

7.

a) When comparing signed with unsigned, the compiler converts the signed value to unsigned. On line 16, we compared num_bytes, which is signed, to buffer_size, which is unsigned.

b) num_bytes_str
- represents invalid number
- represents negative number
- represents numbers in the range [0, 5]
- represents numbers in the range (5, 10]
- represents numbers greater than 10

buffer_size
- is zero
- is in the range [0, 10]
- is greater than 10

Combinations that trigger bug: (negative number, <any>)

c) Base choice: (negative, zero)
Other TRs: (invalid, zero), (0-5, zero), (5-10, zero), (>10, zero),
(negative, 0-10), (negative, >10)

8A

```
@Test
public void frob1() {
        Product product = createProduct();
        Invoice invoice = createInvoice(product);

        List<LineItem> lineItems = invoice.getLineItems();
        assertEquals(lineItems.size(), 1);
}

@Test
public void frob2() {
        Product product = createProduct();
        Invoice invoice  = Mockito.mock(Invoice.class);

        List<LineItem> items = listOf(LineItem(product, invoice, 5, new BigDecimal("69.96"));
        Mockito.mock(invoice.getLineItems(), items);

        List<LineItem> lineItems = invoice.getLineItems();
        actual = lineItems.get(0);

        assertEquals(actual.getInvoice(), invoice);
        assertEquals(actual.getProduct(), product);
        assertEquals(actual.getQuantity(), 5);
        assertEquals(actual.getPrice(), new BigDecimal("69.96"));
}
```

The new tests split the single frob test into 2. This allows us to test individual functionality separately, instead of all in one big test, which makes it easier to discern where bugs originate from and makes the code cleaner.

The new tests also mock the invoice class, which allows us to simulate the getLineItems functionality. The parameters for quantity and price were never specified and magically appeared in the original test (probably default values), here we specify them.

8B

a) Note: can vec even be null? The partition is complete. A vector must either be null, empty, or have at least one element. The partition is disjoint. vec cannot be both null and empty at the same time. vec cannot be both null and contain elements. vec cannot be both empty and contain elements.
b) A set of triples where every block is paired with every other block in some test case, I'm not going to enumerate them here.

Infeasible
- cases containing both B1 and B4
- cases containing both B2 and B4
- cases containing both B6 and B4