Question 1. [8 MARKS]

In this question, there is a 1-mark penalty for each wrong answer. If you leave a part blank, there is no penalty. Your score will not be negative: there is a minimum of 0 on the whole question.

Consider these functions:

```
def fact(n):
    '''Return n!'''

    if n == 0:
        return 1
    else:
        return n * fact(n - 1)


def fib(n):
    '''Return the nth Fibonacci number.'''

    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

Part (a) [2 MARKS]

If `fact(2)` is called, how many calls to `fact` will be on the call stack when `n` is `0`? Check the box next to the correct answer.

Answer: fact(2) calls fact(1) which calls fact(0), so 3 is the correct answer.

Part (b) [2 MARKS]

If `fib(2)` is called, how many calls to `fib` will be on the call stack when `n` is `0`? Check the box next to the correct answer.

Answer: fib(2) calls fib(1), then fib(1) returns, and then fib(0) is called. So 2 is the correct answer.

Part (c) [2 MARKS]

If `fib(3)` is called, how many times is the base case reached? Check the box next to the correct answer.

Answer: Here is a tree of calls:

```
      fib(3)
     /      \
   fib(2)   fib(1)
   /    \
fib(1)  fib(0)
```

There are three calls that do not have a recursive call, so the correct answer is 3.

Part (d) [2 MARKS]

If `fib(3)` is called, how many calls to `fib` are there in total? Check the box next to the correct answer.

Answer: Here is a tree of calls:

```
      fib(3)
     /      \
   fib(2)   fib(1)
   /    \
fib(1)  fib(0)
```

The tree of calls shows that the answer is 5.

Question 2. [5 MARKS]

Complete the following function without using a loop. You can use `len`, but no other built-in functions are allowed. For example, `min` and `max` are *not* allowed.

You are, of course, allowed to index into lists and take slices.

```
def minimum(L):
    '''L is a non-empty list of ints. Return the smallest int in L. L is not modified.'''
```

Answer:

```
    if len(L) == 1:
        return L[0]
    else:
        m = minimum(L[1:])
        if L[0] < m:
            return L[0]
        else:
            return m
```

Question 3. [10 MARKS]

Consider this program.

```
import unittest

class Queue(object):
    def __init__(self):
        '''Make a new empty queue.'''
        self.queue = []

    def enqueue(self, o):
        '''Put o at the end of this queue.'''
        self.queue.append(o)

    def dequeue(self):
        '''Remove and return the front item.'''
        return self.queue.pop(0)

    def front(self):
        '''Return the front item.'''
        return self.queue[0]

    def is_empty(self):
        '''Return whether there are any items in this queue.'''
        return self.queue == []

class TestQueue(unittest.TestCase):

    def setUp(self):
        print "set up Q"
        self.queue = Queue()

    def tearDown(self):
        print "tear down Q"
        self.queue = None

    def test_1(self):
        print "Test 1"
        self.assertTrue(self.queue.is_empty(), "Queue should have been empty.")

    def test_2(self):
        print "Test 2"
        self.queue.enqueue('a')
        self.assertFalse(self.queue.is_empty(), "Queue should not have been empty.")

if __name__ == '__main__':
    unittest.main()
```

Part (a) [2 MARKS]

Check the box next to the true statement. In this Part, there is a 1-mark penalty for a wrong answer. Your score will not be negative: there is a minimum of 0 on the question as a whole.

Answer: Both `test_1` and `test_2` pass.

Part (b) [4 MARKS] What is printed by the program? (Write the output of the `print` statements.)

Answer:

```
set up Q
Test 1
tear down Q
set up Q
Test 2
tear down Q
```

Note that setUp is called before each test method, and tearDown is called after each one. This lets us re-create structures (such as an empty Queue) before each test case so that the test is unaffected by problems in other tests.

Part (c) [4 MARKS]

Write a test method for class `TestQueue` that tests whether two items put into the queue come out in the right order.

Answer:

```
    def test_3(self):
    self.queue.enqueue('a')
    self.queue.enqueue('b')
    self.assertTrue('a' == self.queue.dequeue(), "'a' should have come out first.")
    self.assertTrue('b' == self.queue.dequeue(), "'b' should have come out second.")
```

Question 4. [8 MARKS]

Complete function `count_exceptions`.

```
class ExcA(Exception):
    pass

def count_exceptions(f, L):
    '''Call f, a function that has one parameter, on every item in L and
    return a list of length 3 where:

    the item at index 0 is how many times any exception other than ExcA was raised,
    the item at index 1 is how many times an ExcA exception was raised,
    the item at index 2 is the number of times that f did not raise an exception.
    '''

    # The resulting list.
    resL = [0, 0, 0]
```

Answer:

```
    for item in L:
        try:
            f(item)
            resL[2] += 1
        except ExcA:
            resL[1] += 1
        except Exception:
            resL[0] += 1

    return resL
```

We need to check for ExcA first; if we did the except clauses in the other order, the `except Exception` one would catch the ExcA exceptions.