Graded Exercise 2

CSC140 Foundations of Computer Science

28 February 2020

Today's Graded Exercise has 12 questions.

1. Which of the following statements describes how software engineers test software?

   (a) They continue repeating a test after removing a defect that caused the test to fail.

   (b) They demonstrate that the program returns correct outputs with all possible inputs.

   (c) They prove the complete absence of defects.

   (d) They verify that the program computes all numerical results with perfect precision.

- (a) They continue repeating a test after removing a defect that caused the test to fail.

2. You read about the life of John Aaron, Don Eyles, Margaret Hamilton, or Katherine Johnson. Which one of these lessons might we draw from their examples?

   (a) Title and rank determine who exercises responsibility in any organization.

   (b) Courage, initiative, curiosity, and effort count for more than a person's age or the kind of degree that a person possesses.

   (c) Leadership in a technical field requires deep study of that same field in a university.

(d) Years of work must precede significant achievement.

- (b) Courage, initiative, curiosity, and effort count for more than a person's age or the kind of degree that a person possesses.

3. Examine this code:

```
def factorial( n ):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial( n - 1 )

print( factorial(5) )
```

(a) What will this code print?

(b) Which essential features of a recursive function do you see in this function?

(c) What will happen if this function is called with an argument that is a negative integer?

(a) The program will print "120."

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 120 \end{aligned}$$

(b) The function calls itself (the name of the function appears within the definition of the function) and it contains an **if** statement (there is code that tells the computer when to stop calling the function).

(c) The function will continue to call itself forever if given a negative argument because each successive call gives the function a still smaller (more negative) argument—the argument will never equal 0 or 1 (the stopping conditions).

In practice, the computer will stop executing the program when the program has exhausted the amount of memory that the operating system has allocated for managing function calls.

4. What does this code print?

```
def power( base, exponent ):
    product = 1
    for i in range(0, exponent):
        product *= base
    return product

print( power(2, 0) )
print( power(2, 1) )
print( power(2, 2) )
print( power(2, 6) )
```

The program will print:

```
 1
 2
 4
64
```

This is because:

$$2^0 = 1$$
$$2^1 = 2$$
$$2^2 = 4$$
$$2^6 = 64$$

5. Why might we prefer this function over the previous one?

```
def power( base, exponent ):
    if exponent == 0:
        return 1
    elif exponent == 1:
        return base
    else:
        smaller_power = power_recurse( base, exponent // 2 )
        if exponent % 2 == 0:
            return smaller_power * smaller_power
        else:
            return base * smaller_power * smaller_power
```

This is a recursive function—the function calls itself. The previous function was iterative—it used a loop to build a product.

In this case, recursion makes it possible to raise a number to a power with fewer multiplications than were used in the iterative version of the function.

In this case, the recursive version of the function is more efficient than is the iterative version of the function—it gets the job done with less work.

6. What does this code print?

```
triangular = [ (n * (n + 1))//2 for n in range(8) ]

print( triangular )
```

```
[ 0, 1, 3, 6, 10, 15, 21, 28 ]
```

7. Write code that defines a function that computes T_n, the n^{th} triangular number.

$$T_n = \frac{n \cdot (n+1)}{2}$$

```
def triangular( n ):
    return (n * (n + 1)) // 2
```

8. What does this code print?

```
def f( values ):
    best_guess_so_far = values[0]

    for i in range(1, len(values) ):
        if values[i] > best_guess_so_far:
            best_guess_so_far = values[i]

    return best_guess_so_far

def g( values ):
    best_guess_so_far = 0

    for i in range(1, len(values) ):
        if values[i] > values[best_guess_so_far]:
            best_guess_so_far = i

    return best_guess_so_far

data = [ 3, 1, 4, 1, 5, 9, 2 ]

print( f(data) )
print( g(data) )
```

The program prints:

```
9
5
```

This is because:

- The function f() returns the value of the largest number in a list.
- The function g() returns the index of the largest number in a list.

- In this case, the largest number is 9. It is the sixth number in the list, and so has index 5.

9. There are searches within the selection and insertion sort algorithms.

   (a) In which of the two sorting algorithms is the search through the sorted part of the list?

   (b) In which of the two sorting algorithms is the search through the unsorted part of the list?

   (c) In which of the two sorting algorithms is the search from left to right (toward the last element of the list)?

   (d) In which of the two sorting algorithms is the search from right to left (toward the first element of the list)?

   (e) In which of the two sorting algorithms does the search always reach one end of the list (the first or last element)?

   (f) In which of the two sorting algorithms does the search sometimes stop before reaching one end of the list?

   (a) The insertion sort searches through the sorted part of the list.

   (b) The selection sort searches through the unsorted part of the list.

   (c) The selection sort searches from left to right.

   (d) The insertion sort searches from right to left.

   (e) The search in the selection sort always goes all of the way to the last element in the list.

   (f) The search in the insertion sort does not always go all of the way to the first element in the list.

10. Write the loop that is missing from this function.

```
def merge( a, b ):
    # a is a sorted list of numbers
    # b is a sorted list of numbers
    #
    # return a sorted list that contains
    # all of the numbers in a and b
```

```
    i = 0
    j = 0
    result = []

    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            result.append( a[i] )
            i += 1
        else:
            result.append( b[j] )
            j += 1

    while i < len(a):
        result.append( a[i] )
        i += 1

    # TO-DO add one more loop here

    return result
```

```
    while j < len(b):
        result.append( b[j] )
        j += 1
```

This loop is needed in case the first loop in the function reaches the end of list *a* before it reaches the end of list *b*. In that case, this loop will add the elements that remain in list *b* to list *result*.

11. What does this code print?

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x:6.2f},{self.y:6.2f})"

if __name__ == "__main__":
    u = Vector( 2, 3 )
    v = Vector( 5, 7 )
```

```
print( "u = ", u )
print( "v = ", v )
```

The program prints:

```
(2, 3)
(5, 7)
```

12. The sum of two vectors is another vector. The rule for adding 2 vectors is simple—the x component of the sum is just the sum of the x components of the two vectors, and the y component of the sum is the sum of the y components of the two vectors.

    Here is an example:

$$\begin{aligned} \vec{u} &= (2,3) \\ \vec{v} &= (5,7) \\ \vec{u}+\vec{v} &= (2+5, 3+7) \\ &= (7,10) \end{aligned}$$

    (a) Add to the Vector class a method that adds one vector to another.

    (b) Add to the Vector class program that tests the new method.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # answer to part (a)
    def add(self, other_vector):
        sum_x = self.x + other_vector.x
        sum_y = self.y + other_vector.y
        return Vector( sum_x, sum_y )
```

```
    def __str__(self):
        return f"({self.x:6.2f},{self.y:6.2f})"

if __name__ == "__main__":
    u = Vector( 2, 3 )
    v = Vector( 5, 7 )

    print( "u = ", u )
    print( "v = ", v )

    # answer to part (b)
    sum = u.add( v )
    print( "sum = ", sum )
```

This next exercise is not part of today's examination.

1. Write a program that draws the T-Square fractal, the Sierpinski Gasket, or the Sierpinski Carpet. Look on the Internet for the information that you need.

   Work in a team. Walk around the laboratory. See what other teams are doing.

   Make good use of the time that is available to us. Get as much work done today as you can.

2. Read "Robert Noyce and His Congregation" and "No Silver Bullet: Essence and Accidents of Software Engineering." (You will find links to these articles on Moodle.)

   Take enough notes so that you can contribute to a discussion on Monday.