CS 111: Final Exam Extra Practice Solutions

Question #1

Part A

```
do
no
to
```

Part B

```
17 15 7 7.5
```

Part C

```
3 2
5
4 2
4 3
7
12
```

Here is a table for part C:

range(3, 5) → [3, 4]

i	range(2, i)	j	output	
3	[2]	2	3 2	
		none left	5	exit inner loop, `print(i+j)`
4	[2, 3]	2	4 2	
		3	4 3	
		none left	7	exit inner loop, `print(i+j)`
none left			12	exit outer loop, `print(i*j)`

Part D

```
10 0
7 3
```

Here are some tables for part D:

global

a	b
7	3

foo

a	b
3	7
4	6
5	5
6	4
7	3
8	2
9	1
10	0

Notes:

1) The function call is **`foo(b, a)`**, and thus:
   - **`foo`**'s **`a`** gets a copy of the global **`b`**
   - **`foo`**'s **`b`** gets a copy of the global **`a`**

2) There are two different sets of variables -- one in the global scope, and one belonging to **`foo`**, and any changes that **`foo`** makes to its variables do not change the variables in the global scope.

Question #2

Part A

```
def is_prime(n):
    for i in range(2, n//2+1):
        if n % i == 0:
            return False
    return True
```

Part B

```
def add_primes(lst):
    if len(lst) == 0:
        return 0
    else:
        sum_rest = add_primes(lst[1: ])
        if is_prime(lst[0]):
            return lst[0] + sum_rest
        else:
            return sum_rest
```

If we hadn't required recursion, you could also have used a list comprehension here:

```
def add_primes(lst):
    lc = [ x for x in lst if is_prime(x) ]  # get all primes!
    return sum(lc)
```

Question #3

15 times:
fib(0) and fib(1) are single calls
fib(2) == fib(1) + fib(0), means fib(2) creates 3 total calls (1 + 1 + the original call to fib(2))
fib(3) == fib(2) + fib(1), means fib(3) creates 5 total calls (3 + 1 + the original call to fib(3))
fib(4) == fib(3) + fib(2), means fib(4) creates 9 total calls (5 + 3 + the original call to fib(4))
fib(5) == fib(4) + fib(3), means fib(5) creates 15 total calls (9 + 5 + the original call to fib(5))

Question #4

```
def uniquify(lst):
    if len(lst) == 0:
        return []
    else:
        rest = uniquify(lst[1:])
        if lst[0] in lst[1:]:
            return rest
        else:
            return [lst[0]] + rest
```

Question #5

```
def merge(list1, list2):
    if list1 == []:
        return list2
    elif list2 == []:
        return list1
    else:
        if list1[0] < list2[0]:
            return [list1[0]] + merge(list1[1:], list2)
        else:
            return [list2[0]] + merge(list1, list2[1:])
```

Question #6

Part A

This program calculates how many factors between 1 and n (inclusive) the input number has.
After lines 7, 8 and 9, r3 will only be 0 if r1 was evenly divisible by r2 (i.e., if r1 % r2 == 0), and this will cause us to increment r9, which is the count of the number of factors.
For example, let's trace through an input of 6:

```
line executed               r1   r2   r3   r9
00  read r1                 6
01  setn r9 0               6              0
02  copy r2 r1              6    6         0
03  nop
04  nop
05  nop
06  jeqz r2 14              # r2 != 0, so don't jump
07  div r3 r1 r2            6    6    1    0
08  mul r3 r2 r3            6    6    6    0
09  sub r3 r1 r3            6    6    0    0
10  jgtz r3 12              # r3 == 0, so don't jump
11  addn r9 1               6    6    0    1
12  addn r2 -1              6    5    0    1
13  jumpn 06                # unconditional jump to line 6

06  jeqz r2 14              # r2 != 0, so don't jump
07  div r3 r1 r2            6    5    1    1
08  mul r3 r2 r3            6    5    5    1
09  sub r3 r1 r3            6    5    1    1
10  jgtz r3 12              # r3 > 0, so jump to line 12
12  addn r2 -1              6    4    1    1
13  jumpn 06                # unconditional jump to line 6

06  jeqz r2 14              # r2 != 0, so don't jump
07  div r3 r1 r2            6    4    1    1
08  mul r3 r2 r3            6    4    4    1
09  sub r3 r1 r3            6    4    2    1
10  jgtz r3 12              # r3 > 0, so jump to line 12
12  addn r2 -1              6    3    2    1
13  jumpn 06                # unconditional jump to line 6
```

```
line executed                r1   r2   r3   r9
06  jeqz r2 14               # r2 != 0, so don't jump
07  div r3 r1 r2             6    3    2    1
08  mul r3 r2 r3             6    3    6    1
09  sub r3 r1 r3             6    3    0    1
10  jgtz r3 12               # r3 == 0, so don't jump
11  addn r9 1                6    3    0    2
12  addn r2 -1               6    2    0    2
13  jumpn 06                 # unconditional jump to line 6

06  jeqz r2 14               # r2 != 0, so don't jump
07  div r3 r1 r2             6    2    3    2
08  mul r3 r2 r3             6    2    6    2
09  sub r3 r1 r3             6    2    0    2
10  jgtz r3 12               # r3 == 0, so don't jump
11  addn r9 1                6    2    0    3
12  addn r2 -1               6    1    0    3
13  jumpn 06                 # unconditional jump to line 6

06  jeqz r2 14               # r2 != 0, so don't jump
07  div r3 r1 r2             6    1    6    3
08  mul r3 r2 r3             6    1    6    3
09  sub r3 r1 r3             6    1    0    3
10  jgtz r3 12               # r3 == 0, so don't jump
11  addn r9 1                6    1    0    4
12  addn r2 -1               6    0    0    4
13  jumpn 06                 # unconditional jump to line 6

06  jeqz r2 14               # r2 == 0, so jump to line 14
14  write r9                 # write 4 for the 4 factors (6,3,2,1)
15  halt
```

<u>Part B</u>

```
14 setn r4 2
15 sub r9 r9 r4       # subtract 2 from the number of factors
16 jeqz r9 19         # if get 0, there were only 2 factors, so prime
17 write 1            # if not 0, there were > 2, so write 1 (composite)
18 jumpn 20           # could also have a halt here
19 write 0            # write 0 (prime)
20 halt
```

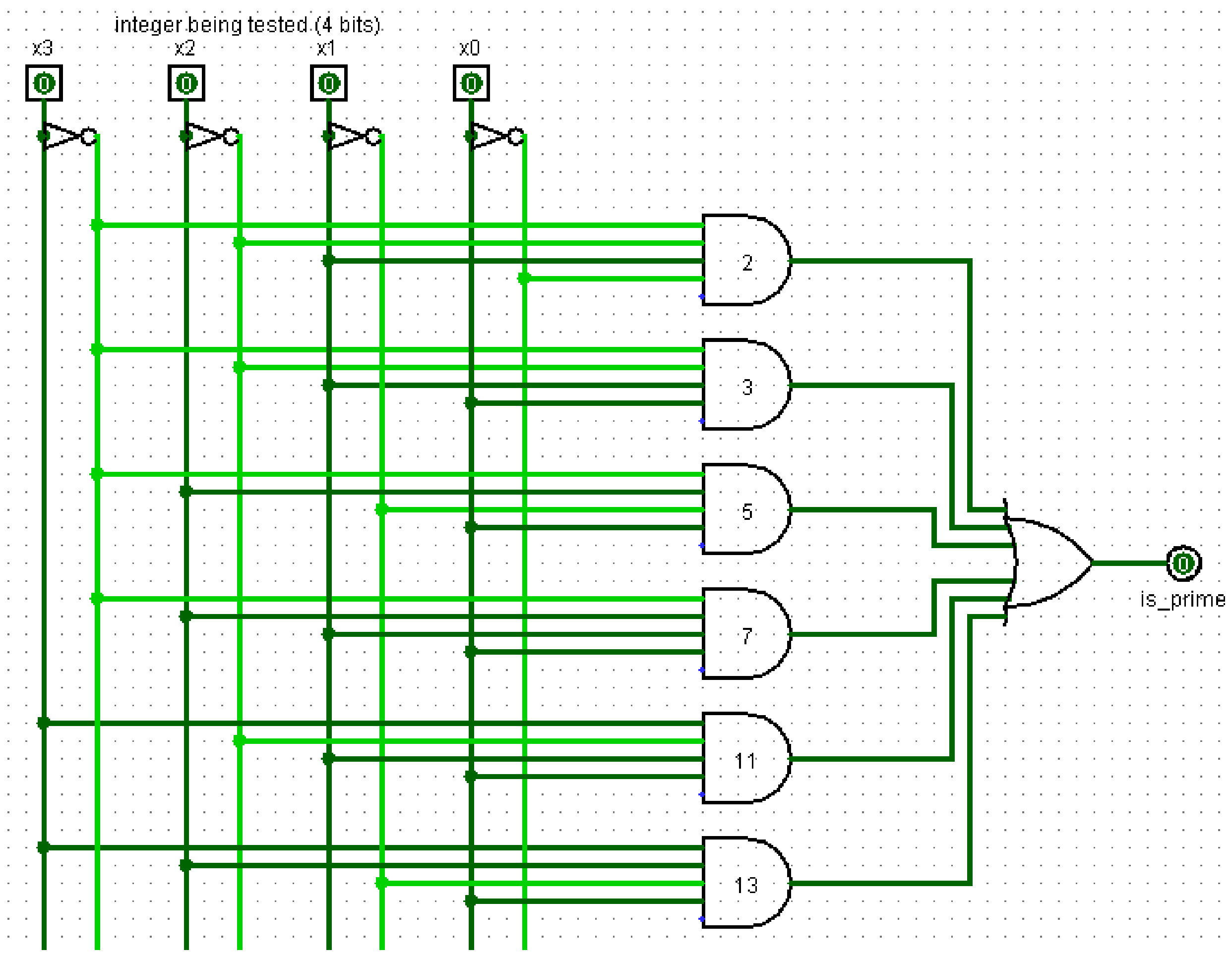
Question #7

Part A

x3	x2	x1	x0	is_prime
0	0	0	0	0
0	0	0	1	0
0	**0**	**1**	**0**	**1**
0	**0**	**1**	**1**	**1**
0	1	0	0	0
0	**1**	**0**	**1**	**1**
0	1	1	0	0
0	**1**	**1**	**1**	**1**
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	**0**	**1**	**1**	**1**
1	1	0	0	0
1	**1**	**0**	**1**	**1**
1	1	1	0	0
1	1	1	1	0

Part B

Each row of the truth table that has an output of 1 gets an AND gate, and those AND gates are then ORed together to produce the final output:



Question #8

```
# loop-based approach
def symmetric(grid):
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] != grid[c][r]:
                return False
    return True

# recursive approach (optional)
def symmetric(grid):
    if len(grid) <= 1:       # first base case
        return True
    top_row = grid[0]
    left_col = [ grid[r][0] for r in range(len(grid)) ]
    if top_row != left_col:  # second base case
        return False
    rest_of_grid = [ grid[r][1:] for r in range(1, len(grid)) ]
    return symmetric(rest_of_grid)
```

Question #9

```
def max(self, other):
    minrows = min(self.nrows, other.nrows)
    mincols = min(self.ncols, other.ncols)
    maxmat = Matrix(minrows, mincols)

    for r in range(minrows):
        for c in range(mincols):
            maxmat.data[r][c] = max(self.data[r][c], other.data[r][c])

    return maxmat
```

Question #10