Second Exam
CS 101 Computer Science I

KEY
Tuesday November 12, 2013
Instructor Muller
Boston College
Fall 2013

Before reading further, please arrange to have an empty seat on either side of you. Now that you are seated, please write your name at the top of this exam.

This is a closed-book and closed-notes exam. Computers, calculators, books and notes are prohibited. In solving problems, feel free to use any function for an earlier problem in a solution for a later problem **even if you didn't finish the earlier one.** For problems other than 1b involving repetition, you are free to use any repetition form that you would like.

Partial credit will be given so be sure to show your work. **Please try to write neatly.**

Problem	Points	Out Of
1		6
2		3
3		3
4		3
5		4
6		3
7		6
		2 (extra credit)
Total		28

1. (6 Points Total) Consider the following (somewhat oddly spaced) rendering of our tail-recursive definition of the `Fibonacci` function:

```
def fibo( n ):

  def loop( n , a , b ):

    if n == 1:

      return a

    else:

      return loop( n - 1, b , a + b )

  return loop( n , 1, 1)
```

   (a) (2 points) Circle all non-parameter occurrences of variables. For each of these, draw an arrow to their governing *formal parameter occurrence*.

   (b) (4 points) Rewrite the `Fibonacci` function using either a `while`-loop or a `for`-loop rather than recursion.

   Answer:

```
def fibo(n):
  a = 1
  b = 1
  while (n > 1):
    c = a + b
    a = b
    b = c
    n = n - 1
  return a
```

```
def fibo(n):
  nums = [1] * n
  if n > 2:
    for i in range(2, n):
      nums[i] = nums[i - 1] + nums[i - 2]
  return nums[n - 1]
```

2. (3 Points) Write a Python function **reverse : a list → a list** such that a call **reverse(myList)** will return the reversal of **myList**. For example, the call **reverse([1, 1, 0, 0])** would return the list **[0, 0, 1, 1]**. Solve this problem without using the built-in mutating reverse function.

Answer:

```
def reverse(xs):
  N = len(xs)
  return [ xs[N - i - 1] for i in range(N)]
```

3. (3 Points) Write a function **reverseRows : (a list) list → (a list) list** such that a call **reverseRows(twoDList)** returns the 2D list resulting from reversing every row in **twoDList**. For example, if **twoDList** is the list:

```
[[1, 2, 3, 4],
 [5, 6, 7, 8]]
```

then the call **reverseRow(twoDList)** would return the 2D list

```
[[4, 3, 2, 1],
 [8, 7, 6, 5]]
```

Answer:

```
def reverseRows(twoDList): return map(reverse, twoDList)
```

4. (3 Points) In lecture, we discussed the positional numeral system and saw, as Leibniz first observed, that the arithmetic algorithms that we learned as children work for numerals expressed in bases other than decimal. We were primarily interested in base 2 (binary) but also in base 16 (hexadecimal or hex). This problem relates to the addition of two bits and a carry-in bit. (This problem isn't concerned with the overall addition algorithm.)

```
          0  <- carry line
    0 1 0 1
  + 0 0 1 1
  ----------
             <- sum line
```

Write a Python function **addBits : (int * int * int) → (int * int)** such that a call **addBits(bit1, bit2, carryIn)** would return the pair **(carryOut, sumBit)** where **sumBit** is the sum bit (i.e., the one that we would write below the line) and **carryOut** is the bit that would carry to the left. Pronouncing the symbol `=>` as "should return", a complete specification of the input/output behavior of **addBits** is as follows:

```
addBits(0, 0, 0) => (0, 0)
addBits(0, 0, 1) => (0, 1)
addBits(0, 1, 0) => (0, 1)
addBits(0, 1, 1) => (1, 0)
addBits(1, 0, 0) => (0, 1)
addBits(1, 0, 1) => (1, 0)
addBits(1, 1, 0) => (1, 0)
addBits(1, 1, 1) => (1, 1)
```

Answer:

```
def addBits(bit1, bit2, carryIn):
  sum = bit1 + bit2 + carryIn
  return (sum / 2, sum % 2)
```

This works well enough, but it somehow seems wrong to define something simpler (i.e., adding two bits) using something more complicated (i.e., division and remainder).

In practice, addition of bits is usually specified using *logical values* `True` and `False` rather than binary numerals. We can think of 1 as `True` and 0 as `False` or we can look at them the other way around. In either case, we can specify the `addBits` functions using the simple logical operators `and`, `or` and `xor` or *exclusive or*.

Table 1: Truth Tables

A	B	A and B	A or B	A xor B
False	False	False	False	False
False	True	False	True	True
True	False	False	True	True
True	True	True	True	False

These operators are often depicted graphically using the following symbols for and, or and xor (left to right):

We can draw a box around this diagram and then duplicate it.

skip

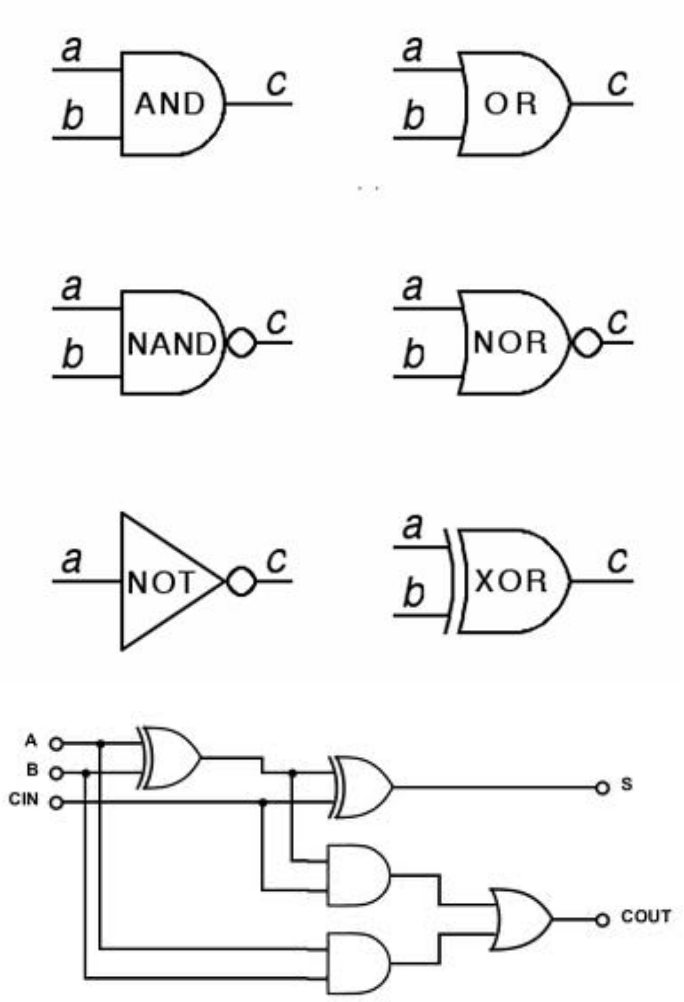


Figure 1: A One-bit Ripple/Carry Adder

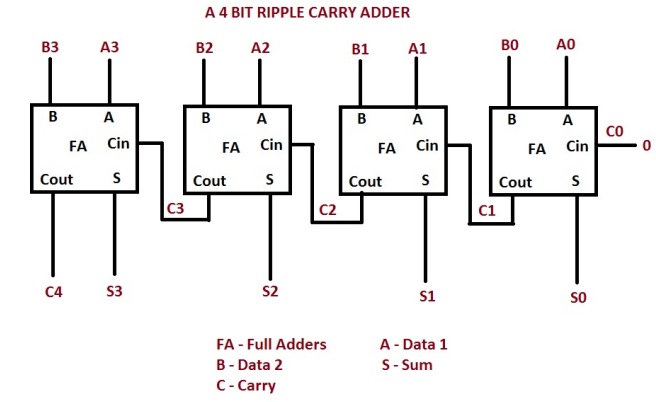


Figure 2: A Four-bit Ripple/Carry Adder

5. (4 Points) Consider the Python function **digits**:

   `def digits(m, n): return (m / n, m % n)`

   and it's application in the process of converting a decimal numeral to another base. For example, to

convert the decimal numeral **13** to binary we could compute as follows:

```
digits(13, 2) = (6, 1)
digits( 6, 2) = (3, 0)
digits( 3, 2) = (1, 1)
digits( 1, 2) = (0, 1)    <- we stop because the quotient is 0
```

Reading the rightmost digits in the four lines above (i.e., the remainders) from bottom to top gives **1101** which is 13 in binary. This method can be used to convert from decimal to any base greater than 1.

Write a Python function **convert : int * int → int list** such that a call **convert(decimalNumeral, newBase)** will return a list of the digits for the numeral in the new base. The call for the example above would be:

```
>>> convert(13, 2)
[1, 1, 0, 1]
```

For the purposes of this problem, you may assume that **newBase** is greater than 1 and less than 10. Of course you may use the **digits** function from above.

Answer:

```
def convert(numeral, base):
  (quotient, remainder) = digits(numeral, base)
  if quotient == 0:
    return [ remainder ]
  else:
    return convert(quotient, base) + [ remainder ]
```

6. (3 Points) Define a Python function **sum2DList : (int list) list → int list** such that a call **sum2DList(a)** returns a list of the sums of each row in the array. For example, if **a** is the array

```
[[1, 2, 3, 4],
 [5, 6, 7, 8]]
```

then the call **sum2DList(a)** would return the list [**10, 26**].

One point extra credit for writing sum2DList without using the built-in sum function.

Answer:

```
def sum2DList(a): return map(sum, a)
```

Or for 1 point extra credit:

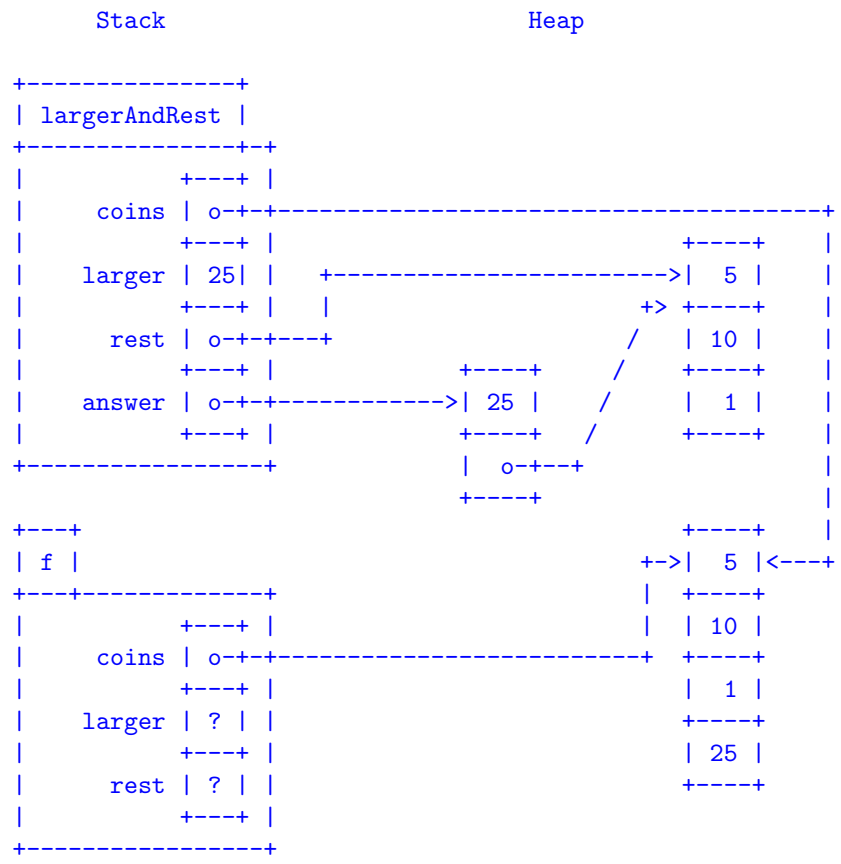
```
def sum2DList(a):
  return map(lambda row : reduce(operator.add, row, 0), a)
```

7. (6 Points Total) Storage Diagrams

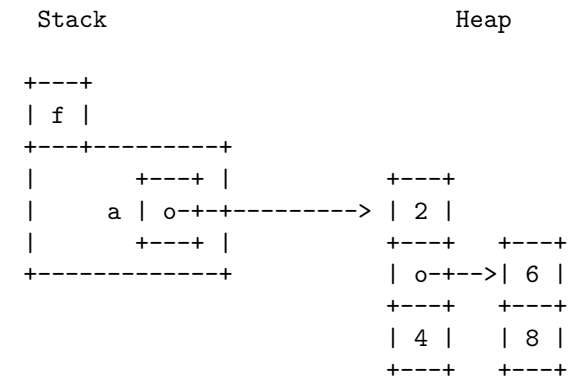
(a) (3 Points) Show the state of the stack and heap after line (1) has been executed but before line (2) has been executed.

```
def largerAndRest(coins):
  larger = max(coins)
  rest = [ coin for coin in coins if coin != larger ]
  answer = (larger, rest)                                    (1)
  return answer                                              (2)

def f():
  coins = [5, 10, 1, 25]
  (larger, rest) = largerAndRest(coins)

f()
```

Answer:



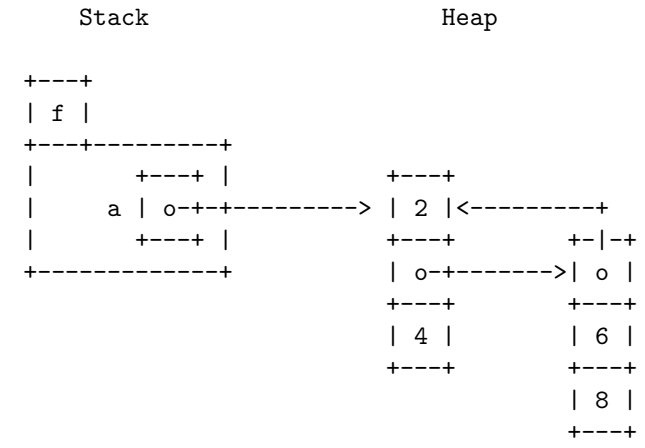
(b) (3 Points) Write Python code that would give rise to the following storage diagram. Annotate the code with the label (1) at the point in the code where the diagram would exist in memory.



Answer:

```
def f():
  a = (2, (6, 8), 4)                    (1)
  return a

f()
```

(c) (1 Point Extra Credit) Write Python code that would give rise to the following storage diagram. Annotate the code with the label (1) at the point in the code where the diagram would exist in memory.



Answer:

```
def f():
  a = [2, [0, 6, 8], 4]
  a[1][0] = a                               (1)
  return

f()
```