cs1120: Exam 1

Due: Thursday, 11 October at 3:30pm (in class)

UVA ID (e.g., wrw6y) : ANSWER KEY

Directions

Work alone. You may not discuss these problems or anything related to the material covered by this exam with anyone except for the course staff between receiving this exam and turning it in.

Open resources. You may use any books you want, lecture notes, slides, your notes, and problem sets. You may ***not*** use PyCharm or any Python or Udacity Python, but it is not necessary to do so. You may also use external non-human sources including books and web sites. If you use anything other than the course books, slides, and notes, cite what you used. You may not obtain any help from other humans other than the course staff.

Answer well. Answer all questions 1-9 (question 0 is your UVA ID in two places and turning in a stapled, printed exam, which hopefully everyone will receive full credit for), and optionally answer questions 10-11.

You may either: (1) print out this exam and write your answers on it or (2) write your answers directly into the provided Word template and print the result out. Whichever one you choose, you must turn in your answers printed **on paper** and they must be clear enough for us to read and understand. You should not need more space than is provided to write good answers, but if you want more space you may attach extra sheets. If you do, make sure they are clearly marked.

The questions are not necessarily in order of increasing difficulty, so if you get stuck on one question you should continue on to the next question. There is no time limit on this exam, but it should not take a well-prepared student more than a few hours to complete. It may take you longer, though, so please do not delay starting the exam. There is no valid excuse (other than a medical or personal emergency) for running out of time on this exam.

No "snow jobs". If you leave a question **blank**, you will receive **three** points for it. If you have no idea and waste our time with long-winded guessing, we will be less sanguine and the grading will be more sanguine. :-)

Use any Python procedure from class. In your answers, you may use any Python procedure that appears in the lecture notes or in the book without redefining it (e.g., len, filter, sort, find_best, etc.). If there are multiple similar names (e.g., len vs. length, map vs. list_map), use whichever you like.

Full credit depends on the clarity and elegance of your answer, not just correctness. Your answers should be as short and simple as possible, but not simpler. Your programs will be judged for correctness, clarity and elegance, but you will not lose points for trivial errors (such as missing a closing parenthesis).

UVA ID *again* (e.g., wrw6y) : ANSWER KEY

Your Scores

0	1	2	3	4	5	6	7	8	9	EC	Total
10	10	10	10	10	10	10	10	10	10	2	100

(Your scores are recorded on the second page so that they are not visible to other students when tests are distributed or passed back.)

1. Consider the following grammar:

S ::= N
N ::= A B C X | D E F X
A ::= edith | simone
B ::= de | ε
C ::= wharton | beauvoir
D ::= percy
E ::= bysshe | ε
F ::= shelley
X ::= and S | ε

The start symbol is S. The symbol ε denotes the empty string.
(a) Is the language of this grammar infinite or finite?
(b) Give a string of length six (i.e., six words) that is in the language of the grammar.

(a) Infinite.
(S can derive "edith wharton and S" in two steps. In addition, S can derive "edith wharton". So there are an infinite number of strings in the language of the grammar.)

(b) edith wharton and percy bysshe shelley
(The trick is that "and" adds one word, so you have to pick one two-word name and one three-word name: 2+1+3 = 6 or 3+1+2=6.)

2. Consider the following Python definition.

```
def strainer(pred, lst):
        if lst == []:
                return []
        elif pred(lst[0]):
                return [lst[0]] + strainer(pred, lst[1:])
        else:
                return [lst[0]]
```

Provide a convincing argument that **`strainer`** is not equivalent to **`filter`**. (Hint: provide inputs on which they behave differently.)

Strainer is not equivalent to filter when any element of the list fails to satisfy the predicate.

The source to filter looks like:

```
def filter(pred, lst):
      if lst == []:
            return []
      elif pred(lst[0]):
            return [lst[0]] + filter(pred, lst[1:])
      else:
            return [] + filter(pred,lst[1:])
```

The differences are bolded and colored. In the final case where the list is not empty and the predicate is not satisfied, strainer returns [lst[0]] instead of the result of a recursive call.

Thus:

```
>>> filter(is_odd, [1,3,5,7,8,9,11])
[1,3,5,7,9,11]

>>> strainer(is_odd, [1,3,5,7,8,9,11])
[1,3,5,7,8]
```

In this example, note how strainer never gets to 9 or 11 because it returns the first element that fails the predicate (8) and then stops!

Since there is one input on which the functions behave differently, they are not equivalent.

3. Write a Python procedure **`censor`** that accepts a list of strings as input. It should return the same list of strings in the same order, but with all instances of "profanity" replaced by "bleep". For example:

```
>>> censor(["with", "these", "profanity", "snakes"])
["with", "these", "bleep", "snakes"]
>>> censor(["this"] + ["profanity", "plane"])
["this", "bleep", "plane"]
```

```
# Basic Answer
def censor(lst):
     if lst == []:
          return []
     elif lst[0] == "profanity":
          return ["bleep"] + censor(lst[1:])
     else:
          return [ lst[0] ] + censor(lst[1:])

# Equivalent, More Concise Answer
def censor(lst):
     if lst == []:
          return []
     return [ "bleep" if lst[0] == "profanity" \
               else lst[0] ] + censor(lst[1:])

# Equivalent Answer Using Map()
def censor(lst):
     return map(lambda x : \
       "bleep" if x == "profanity" else x, lst)

# Equivalent Answer Using List Comprehensions
def censor(lst):
     return ["bleep" if x == "profanity" else x \
             for x in lst]
```

4. Define a `make_list_min_cutoff` procedure that takes one input, a number, and *produces a procedure* as output. The output procedure is a procedure that takes a list of numbers as input, and produces as output that same list but with all entries *less than* the original element removed. (Hint: Review those curve transformations from Problem Set 3.) For example:

```
>>> (make_list_min_cutoff (5))
<function>
>>> (make_list_min_cutoff (3))( [1,2,3,4,5] )
[3, 4, 5]
>>> (make_list_min_cutoff (7))( [-1,8,-3,2] )
[8]
```

```
# Since make_list_min_cutoff() returns a procedure,
# you should either have a nested procedure definition
# and return that procedure or return a lambda
# expression.

# Basic Answer
def make_list_min_cutoff(cutoff):
     def my_filter(lst):
          if lst == []:
               return []
          elif lst[0] < cutoff:
               return my_filter(lst[1:])
          else: # meets cutoff
               return [lst[0]] + my_filter(lst[1:])
     return my_filter

# Equivalent Answer with Filter()
def make_list_min_cutoff(cutoff):
     def my_filter(lst):
          return filter(lambda x : x >= cutoff, lst)
     return my_filter

# Equivalent Answer with Lambda instead of Nested-Def
def make_list_min_cutoff(cutoff):
     return lambda lst: \
               filter(lambda x : x >= cutoff, lst)

# Equivalent Answer with List Comprehensions
def make_list_min_cutoff(cutoff):
     return lambda lst: [x for x in lst if x >= cutoff]
```

5. Define a procedure **`nested_catenate`** that takes one input: a list. The list may contain string elements, but it may also contain other lists (which may themselves contain other lists, hence "nested"). The procedure should return the string concatenation the elements anywhere in the list, in presentation order. For example:

```
>>> isinstance([1,2,3], list)
True
>>> isinstance("hello", list)
False
>>> nested_catenate(["a", "b", "c", "d"])
"abcd"
>>> nested_catenate(["w", ["x", "y"], "z"])
"wxyz"
>>> nested_catenate([])
""
>>> nested_catenate( ["1", ["2", ["3"], "4", []], "5"])
"12345"
```

(Hint 1: `rewrite_lcommands`. Hint 2: There are three cases. The list may be empty, its first element may itself be a list, or its first element may be a string. Two of those three cases involve recursive calls.)

```
# Basic Answer
def nested_catenate(lst):
     if lst == []:
          return "" # Base Case returns Empty String
     elif isinstance(lst[0], list):
          return nested_catenate(lst[0]) + \
                 nested_catenate(lst[1:])
     else:
          return lst[0] + nested_catenate(lst[1:])

# Equivalent Answer with Helper Functions
def flatten(lst): # turns [1,[2,[],[3]],4] -> [1,2,3,4]
     if lst == []:
          return []
     elif isinstance(lst[0], list):
          return flatten(lst[0]) + flatten(lst[1:])
     else:
          return [lst[0]] + flatten(lst[1:])

def catenate(lst): # only works on flattened lists
     if lst == []: return ""
     else: return lst[0] + catenate(lst[1:])

def nested_catenate(lst):
     return catenate(flatten(lst))
```

6. Answer each of the following questions about function growth. Give an argument that explains each answer. For example, to prove that n is in O($n/2$) you might demonstrate the constants $n_0 = 1$ and $c = 2$. If there is any ambiguity, use the definitions from Chapter 7 of the course book.

Recall: g is in O(f) iff there are positive constants c and n0 such that g(n) <= c f(n) for all n >= n0.

Is $2n-8$ in O(n) ? Why or why not?

Yes. Pick c=2 and n0=1. That yields the true equation 2n-8 <= 2n for all n >= 1.

(Could also pick c > 2 or n0 > 1.)

Is n^5 in O(n^4) ? Why or why not?

No. Intuitively, n^5 grows faster than n^4. Formally, suppose there were positive constants c and n0 such that $n^5 <= c\ n^4$ for all n >= n0. Now I just need to find an n >= n0 that makes the equation false (see Slide #14 of Class Notes #09). Since n >= n0 > 0, we can divide by n, simplifying the equation to: n <= c for all n >= n0. So just pick n > c to make the equation false.

Is n in Ω($n/2 + 7$) ? Why or why not?

Yes. Pick c=1 and n0=14. That yields the true equation n >= n/2 + 7 for all n >= 14.

Is log n in Ω(n^4) ? Why or why not?

No. Intuitively, **log** n grows much more slowly than n^4. Formally, suppose there were positive constants c and n0 such that **log** n >= c n^4 for all n >= n0. Now I just need to find an n >= n0 that makes the equation false. Since log n < n for all n > 1 and n < n^4 for all n > 1, by transitivity, log n < n^4 for all n > 1. Since c is a positive constant, log n < c n^4 for all n > 1. So any n greater than 1 and greater than n0 makes the equation false.

Is $5n^2$ in Θ(n^2+n^3) ? Why or why not?

It is not. To be in **Θ** it must be in O and **Ω**. However, $5n^2$ is ***NOT*** in **Ω**(n^2+n^3). Intuitively, n square does not grow faster than n cubed. Formally, suppose there were positive constants c and n0 such that $5n^2$ >= **c**(n^2+n^3). Now I just need to find an n >= n0 that makes the equation false. So let's factor and simplify the equation: $5n^2$ >= **c**n^2(1+n). Since n is positive, it is safe to divide both sides by n^2. So **5** >= **c**(1+n) so just pick any n greater than n0 that also satisfies n > 5/c – 1.

7. Consider the following four procedures:

```
def filter(f, lst):           # Same as in class/notes/book
      if lst == []: return []
      if f(lst[0]): return [lst[0]] + filter(f, lst[1:])
      return filter(f, lst[1:])

def my_reverse(lst):          # Reverse the list
      def reverse_helper(x,y):
            if x == []: return y
            return reverse_helper(x[1:], [x[0]] + y)
      return reverse_helper(lst, [])

def revfilter_alpha(f, lst):  # Reverse and filter ...
      return my_reverse(filter(f, lst))

def revfilter_beta(f, lst):   # Reverse and filter ...
      if lst == []: return []
      return revfilter_beta(f, lst[1:]) + \
            ([lst[0]]  if f(lst[0])  else [])
```

Give the running time of `my_reverse` and `revfilter_alpha` and `revfilter_beta` in Big Theta **Θ** notation. Assume `f` runs in constant time. Which of `revfilter_alpha` and `revfilter_beta` is faster? Can there be an asymptotically faster `revfilter` procedure? (Hint: either demonstrate "yes" by giving the procedure, or argue that no such faster procedure is possible.)

(a) my_reverse is Θ(n) where n is the length of the list.

There are n recursive calls. Each recursive call takes O(1) time because "x[1:]" takes O(1) time and "[x[0]] + y" takes O(1) time (adding one element to the front of a list).

(b) revfilter_alpha is Θ(n) where n is the length of the list. First, the call to filter takes Θ(n) time (assuming that f() runs in constant time). Then the call to my_reverse() takes Θ(n) time. So Θ(n+n) = Θ(n).

(c) revfilter_beta is $\Theta(n^2)$ where n is the length of the list. There n recursive calls. However, each recursive call takes Θ(n) time. To see why, consider the list append on the last line: revfilter_beta() + [one_element]. To add an append an element *to the end* of a list takes Θ(n) time because you have to walk all the way down the list, consider the last element, and then build the result back up. (This was true in part (a) as well, but in that case the first list was one element long, so the time was O(1).) So Θ(n)*Θ(n) $\Theta(n^2)$.

(d) The faster procedure is **revfilter_alpha**. n is in $O(n^2)$.

(e) No, the fastest possible is Θ(n) since you must visit every element at least once (to decide whether to keep it or not) and there are n element.

8. Consider the task of sorting a music playlist for a portable music player or cell phone. We will represent each song as a list of its artist and its title. We wish to sort the playlist by *title first*, and then by artist. Write a procedure `sort_playlist` that accepts one argument, a list of songs, and returns that same list of songs, but sorted as per the description above. Example:

```
>>> song1 = ["ABBA", "Mamma Mia"]
>>> song2 = ["Taylor Swift", "Love Story"]
>>> song3 = ["Beyonce", "Single Ladies"]
>>> song4 = ["Randy Newman", "Love Story"]
>>> song5 = ["Lonely Island", "I'm On A Boat"]
>>> sort_playlist([song1, song2, song3, song4, song5])
[     ["Lonely Island", "I'm On A Boat"],
      ["Randy Newman", "Love Story"],
      ["Taylor Swift", "Love Story"],
      ["ABBA", "Mamma Mia"],
      ["Beyonce", "Single Ladies" ] ]
>>> "ABBA" < "Beyonce"  # < does dictionary ordering on strings
True
>>> "ABBA" == "Beyonce" # check string equality
False
```

(No points off if you sort Z-A instead of A-Z. All points off if you sort by artist first.)

```
# Sort code copied from, for example Slide #06 of Class #09
# You did not need to copy in the source code.
def find_best(things, better):
     if len(things) == 1: return things[0]
     rest = find_best(things[1:], better)
     return things[0] if better(things[0], rest) else rest

def sort(lst, cf):
     if not lst: return []
     best = find_best(lst, cf)
     lst.remove(best) # tricky!
     return [best] + sort(lst, cf)

# The real answer begins:
def sort_playlist(lst):
     # we compare by TITLE first, then by ARTIST
     def song_comes_first(a,b):
          if a[1] < b[1]: return true
          elif a[1] == b[1]: return a[0] < b[0]
          else: return false
     return sort(lst, song_comes_first)
```

9. You should write two procedures. The first, `number_of_friends`, takes as input a lists of pairs of strings (representing friends) and a particular string (the person under consideration) and returns an integer (the number of friends of that person). The second, `most_friends`, takes as input a **non-empty** list of pairs of strings (representing friends, as before). It should return the name of the person with the most friends. (This can be done in under 10 lines total.) Example:

```
>>> friends = [ [ "rachel", "monica" ],   # rachel & monica are friends
                [ "phoebe", "rachel" ],   # as are phoebe & rachel, ...
                [ "rachel", "joey" ],
                [ "phoebe", "joey" ],
                [ "monica", "chandler" ] ]
>>> number_of_friends(friends, "rachel")
3                                         # monica, phoebe, joey
>>> number_of_friends(friends, "chandler")
1                                         # monica
>>> most_friends(friends)
"rachel"
```

```
# Basic Answer: check on left and on right
def number_of_friends(lst, who):
     if lst == []: return 0
     elif (lst[0][0] == who) or (lst[0][1] == who):
          return 1 + number_of_friends(lst[1:], who)
     else: return number_of_friends(lst[1:], who)

# Equivalent Answer
def number_of_friends(lst, who):
     if lst == []: return 0
     return (1 if who in lst[0] else 0) + \
          number_of_friends(lst[1:], who)

# Equivalent Concise Answer
def number_of_friends(lst, who):
     return len(filter(lambda pair: who in pair, lst))

# Basic Answer: Similar to find_best()
def most_friends(lst):
     def pick_more_friends(a,b):
          return a if number_of_friends(lst,a) > \
                      number_of_friends(lst,b) else b
     if len(lst) == 1:
          return pick_more_friends(lst[0][0], lst[0][1])
     return pick_more_friends(lst[0][0],
               pick_more_friends(lst[0][1],
                    most_friends(lst[1:])))
```

```
# Equivalent Answer: gather up all people, find best
def most_friends(lst):
     def more_friends(a,b):
          return number_of_friends(lst,a) >
               number_of_friends(lst,b)
     def all_people_mentioned(lst):
          if lst == []: return []
          else return [lst[0][0]] + [lst[0][1]] +
               all_people_mentioned(lst[1:])
     # Uses "find_best" code from, say, Slide #6 of
     # Class Notes #09
     return find_best(all_people_mentioned(lst),
          more_friends)

# Equivalent Answer: Nested List Comprehensions
# (so intense!)
# I have "most friends" if the list of people with
# more friends than me is []. Find all people with
# "most friends" and return the [0]th.
def most_friends(lst):
  people = [x[0] for x in lst] + [x[1] for x in lst]

  return ([ me for me in people

            if [] == [ better for better in people if
                     number_of_friends(lst,better) >
```

10. (Extra credit, two points max, no points for leaving this blank.) Give the running time for your implementation of `most_friends` using Θ notation (or a hypothetical implementation if you did not complete #9). Assume that the friend list contains *F* friend pairs.

```
As written, my basic most_friends procedure runs in
Theta(F^2) time. There are F recursive calls to
most_friends(). Each recursive call runs in Theta(F)
time because it involves a call to number_of_friends().
The function number_of_friends() runs in Theta(F) time
because there are F recursive calls of O(1) time each
(equivalently, because you must visit every friend-pair
because "who" may be involved in the last one).

So Theta(F) * Theta(F) = Theta(F^2).
```