Last Name: ______________________ First: ______________________ Netid: _________ Section _________

CS 1110 Final, December 16th, 2013

This 150-minute exam has 8 questions worth a total of 100 points. Scan the whole test before starting. Budget your time wisely. Use the back of the pages if you need more space. You may tear the pages apart; we have a stapler at the front of the room.

It is a violation of the Academic Integrity Code to look at any exam other than your own, to look at any other reference material, or to otherwise give or receive unauthorized help.

You will be expected to write Python code on this exam. We recommend that you draw vertical lines to make your indentation clear, as follows:

```
def foo():
|   if something:
|   |   do something
|   |   do more things
|   do something last
```

Unless you are explicitly directed otherwise, you may use anything you have learned in this course.

Question	Points	Score
1	2	
2	14	
3	12	
4	18	
5	14	
6	12	
7	16	
8	12	
Total:	100	

The Important First Question:

1. [2 points] Write your last name, first name, netid, and *lab section* at the top of each page.

Last Name: ______________________ First: ______________________ Netid: _________ Section _________

Throughout this exam, there are several questions on sequences (strings, lists, and tuples). All sequences support slicing. In addition, you may find the following expressions below useful (though not all of them are necessary).

Expression	**Description**
`len(s)`	**Returns**: number of elements in sequence `s`; it can be 0.
`x in s`	**Returns**: `True` if `x` is an element of sequence `s`; `False` otherwise.
`s.index(x)`	**Returns**: index of the FIRST occurrence of `x` in `s`. Raises a `ValueError` if `x` is not found.
`s.rindex(x)`	**Returns**: index of the LAST occurrence of `x` in `s`. Raises a `ValueError` if `x` is not found.
`s.count(x)`	**Returns**: number of times `x` appears in `s`. Could be zero
`x.pop()`	(**Lists Only**) **Returns**: The first element of `x`, removing it from the list.
`x.append(a)`	(**Lists Only**) Adds `a` to the end of list `x`, increasing length by 1.
`x.extend(y)`	(**Lists Only**) Appends each element in `t` to the end of list `x`, in order.
`x.insert(i,y)`	(**Lists Only**) Inserts `y` at position `i` in list `x`. Elements after position `i` are shifted to the right.

In addition, you should remember the typecasting functions (`int`, `float`, `bool`, etc.). These convert a value from one type to another, and raise a `ValueError` if conversion is not possible.

2. [14 points total] **Classes**
The next page contains skeleton code for two classes: `Course` and `Instructor`.

(a) [7 points] Implement the getters and initializers for both these classes as follows:

- The getter for a list attribute should return a copy; other getters act as normal.
- Both initializers have only one additional parameter: the name.
- The precondition for the initializer should match the attribute invariant.
- The attribute invariants should be satisfied when the initializer is complete.

The methods should be fully specified, and preconditions should be enforced with asserts.

(b) [7 points] These classes have a special invariant that link them together: if an instructor has a course in `_list`, that instructor must be in attribute `_prof` of that course, and vice versa. For example, if method `f.removeCourse(c)` is called to remove course `c`, this method should also be sure to make `c._prof None` (by calling `c.removeInstructor()`).

Because attributes are hidden, this means that methods in one class must call methods in another. **You must take care to avoid an infinite set of calls**. You do not want the case where `c.removeInstructor()` calls `self._prof.removeCourse(c)`, which calls `c.removeInstructor()`, which calls `self._prof.removeCourse(c)`, and on and on. Some duplicate calling may be necessary, but it should not be infinite.

Note: This is a trickier problem and you may want to come back to this later.

Last Name: ______________________ First: ______________________ Netid: _________ Section _________

```python
class Course(object):
    """Instance is a course at Cornell.

    INSTANCE ATTRIBUTES:
       _name: course name
              [nonempty str, IMMUTABLE]
       _prof: instructor
              [Instructor or None]

    If _prof is not None, this instance must
    be included in the _list attribute."""

    # GETTERS FOR NAME & INSTRUCTOR
    # DO NOT PUT SETTERS HERE
    def getName(self):
        return self._name

    def getInstructor(self):
        return self._prof


    # INITIALIZER; HAS PARAMETER FOR NAME
    def __init__(self,n):
        assert type(n) == str and n != ''
        self._name = n
        self._prof = None


    # SPECIALIZED SETTERS
    def removeInstructor(self):
        """If this course has instructor
        (_prof not None), remove it"""
        if not self._prof is None:
            prof = self._prof
            self._prof = None
            prof.removeCourse(self)


    def makeInstructor(self,t):
        """Make t this course's instructor.
        If current instructor neither None
        nor t, remove previous instructor.
        Prec: t an Instructor (not None)."""
        assert type(t) == Instructor
        prof = self._prof
        if not prof is None and prof != t:
            self.removeInstructor()
        self._prof = t
        t.addCourse(self)
```

```python
class Instructor(object):
    """Instance is a teacher at Cornell.

    INSTANCE ATTRIBUTES:
       _name: teacher name
              [nonempty str, IMMUTABLE]
       _list: list of courses taught
              [list of Course objects]

    For each element of _list, this instance
    must be set as the current instructor."""

    # GETTERS FOR NAME & COURSE LIST
    # LIST GETTER SHOULD COPY LIST
    def getName(self):
        return self._name

    def getCourses(self):
        return self._list[:]


    # INITIALIZER; HAS PARAMETER FOR NAME
    def __init__(self,n):
        assert type(n) == str and n != ''
        self._name = n
        self._list = []


    # SPECIALIZED SETTERS
    def removeCourse(self,c):
        """Remove c from this instructor's
        course list, if necessary.
        Prec: c a Course (not None)"""
        assert type(c) == Course
        if c in self._list:
            self._list.remove(c)
            c.removeInstructor()


    def addCourse(self,c):
        """If this instructor not teaching
        course c, add it as a new course.
        Prec:  c a Course (not None)"""
        assert type(c) == Course
        if not c in self._list:
            self._list.append(c)
            c.makeInstructor(self)
```

3. [12 points total] **Testing and Error Handling**

   (a) [6 points] Below is the specification of a function demonstrated in class. **Do not implement it**. In the space below,provide at least five different test cases to verify that this function is working correctly. For each test case provide: (1) the function input, (2) the expected output, and (3) an explanation of what makes this test *significantly* different.

```
def ispalindrome_loosely(s):
    """Returns: True if s is a palindrome considering only the letters.
    Case and any non-letter characters are ignored.
    Example: 'A man, a plan, a canal.  Panama!' evaluates to True.
    Precondition: s is a string"""
```

There are several possible answers, but they must all be *significantly* different. Here are some examples of some significantly different tests:

Input	**Output**	**Reason**
`'abc'`	False	Not a palindrome
`'aba'`	True	Simple palindrome
`'abA'`	True	Palindrome, ignoring case
`'a,ba'`	True	Palindrome, ignoring punctuation
`'ab,A'`	True	Palindrome, ignoring both
`''`	True	Empty string is a special palindrome

   (b) [6 points] Implement the function specified below. Your implementation will require that you use a try-except statement.

```
def next_int(s):
    """Returns: Returns string representation of next int after string s.
    If s does not represent an int, it returns None.
    Example: '10' evaluates to '11', while 'a' evaluates to None.
    Precondition: s is a string"""
    # Try block is if no errors occurred.
    try:
        n = int(s)
        n = n+1
        return str(n)

    except:
        pass # Returning here is okay, too

    return None
```

4. [18 points total] **Iteration and Recursion**
A transform is a function that takes a point in one coordinate system and converts it into a point in another coordinate system. Converting from Cartesian coordinates to radial (or spherical) coordinates is a transform. Transforms allow us to use whatever coordinate system makes our work easier. For example, radial coordinates simplified a lot of our animation code in class.

Wavelet analysis, an important tool in signal processing, deals with a large number of transforms. In this problem, you will work with the simplest type of wavelet transform, the Haar transform. This transform processes 2^n dimensional points, which we will represent as 2^n lists of numbers.

(a) [10 points] At each step of the calculation, the Haar transform will halve the size of the original list using two functions: pairwise average and pairwise difference. Pairwise average groups the elements of the list into consecutive pairs and averages them. For example, the pairwise average of `[9, 7, 3, 5]` is `[8,4]`, since $(9+7)/2$ is 8 and $(3+5)/2$ is 4. Pairwise difference is the same, except that it subtracts the pairs. So the pairwise difference of `[9, 7, 3, 5]` is `[1,-1]`, since $(9-7)/2$ is 1 and $(3-5)/2$ is -1.

Implement each of these operations below. Implement `pair_average` **with a for-loop** and implement `pair_difference` **with a while-loop**. You do not need to state invariants.

```
def pair_average(data):
    """Returns: Pairwise average of list of size 2^(n-1).
    Example: [2, 4, 1, 0, 0, -1, -1, 1] evaluates to [3, 0.5, -0.5, 0]
             [3, 2, -1, -2] evaluates to [2.5, -1.5]
    Implementation must use a for-loop.
    Precondition: data is an 2^n element list of numbers."""
    # accumulator
    output = []
    for k in range(len(data)/2):
        average = (data[2*k] + data[2*k+1])/2.0
        output.append(average)
    return output


def pair_difference(data):
    """Returns: Pairwise difference of list of size 2^(n-1).
    Example: [2, 4, 1, 0, 0, -1, -1, 1] evaluates to [-1, 0.5, 0.5, -1]
             [3, 2, -1, -2] evaluates to [0.5, 0.5]
    Implementation must use a while-loop.
    Precondition: data is an 2^n element list of numbers."""
    # accumulator
    output = []
    k = 0 # loop variable
    while k < len(data):
        average = (data[k] - data[k+1])/2.0
        output.append(average)
        k = k + 2
    return output
```

(b) [8 points] The actual Haar transform is a procedure that takes a 2^n list as input, and produces a 2^n list as output. If the list is length 2, it computes both the pairwise average `[a]` and pairwise difference `[d]`. It returns the list `[a,d]` when done.

If the list is of size 2^n where $n > 1$, then it computes the transform recursively:

- Compute the pairwise average of the data.
- Compute the Haar transform of the pairwise average.
- Compute the pairwise difference of the data.
- Concatenate the transformed pairwise average and the difference.

This recursive function is specified below. Implement it.

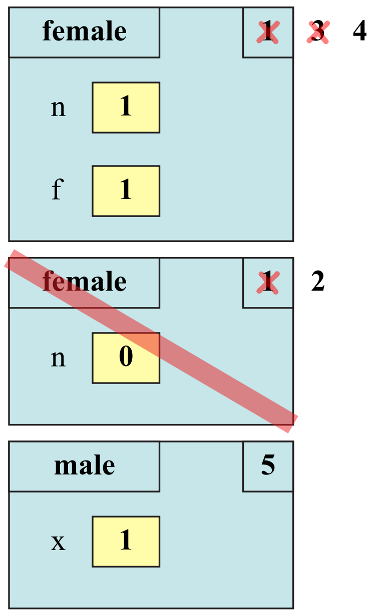
```
def haar_transform(data):
    """Return: the Haar transform of the list.
    Example: To compute the Haar transform of [9, 7, 3, 5]
    1.  Compute the pairwise average, (9+7)/2 = 8, (3+5)/2 = 4, or [8,4]
    2.  Compute the Haar transform of this average (to get [6,2])
    3.  Compute pairwise difference, (9-7)/2 = 1, (3-5)/2 = -1, or [1,-1]
    4.  Append pairwise difference to the transform of the first half.
    The final answer is [6, 2, 1, -1].
    Precondition: data is a 2^n element list of numbers, n >= 1."""

    # Both cases need this.
    first = pair_average(data)
    secnd = pair_difference(data)

    ##### BASE CASE #####
    if len(data) == 2:
        return first+secnd


    ##### RECURSIVE CASE #####
    first = haar_transform(first)
    return first+secnd
```

5. [14 points total] **Call Frames and Memory Diagrams**

(a) [7 points] Douglas Hofstadter, famous for the 1980 Pulitzer Prize winning *Gödel, Escher, Bach: an Eternal Golden Braid*, defined the following two mutually recursive functions:

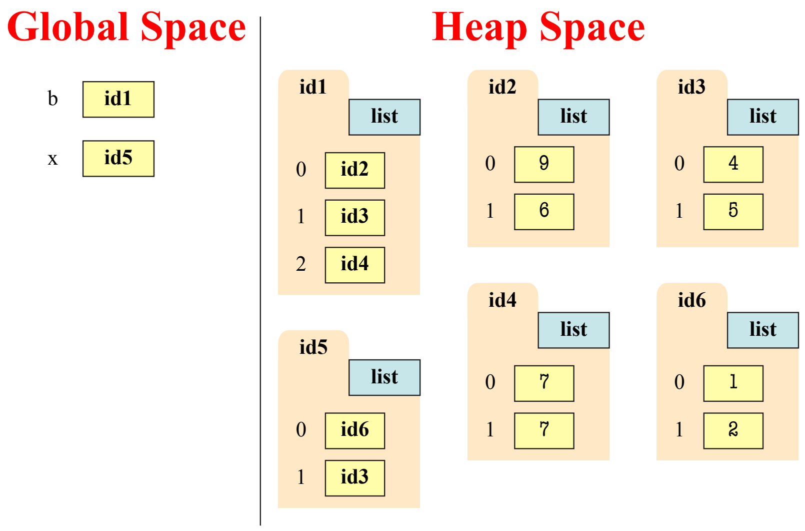
```
def female(n):                          def male(x):
    if (n == 0):             1              if (x == 0):             5
        return 1             2                  return 0             6
    f = female(n-1)          3              m = male(x-1)            7
    return n-male(f)         4              return x-female(m)       8
```

Execute the call `female(1)`, drawing the call stack below. Stop when you have drawn the **third frame and assigned arguments to the parameters**. This is including any frames that you might have erased (e.g. it is not necessarily the case that the call stack has three frames at this time).
Do not draw multiple diagrams for each frame; each frame should only be drawn once. If you are supposed to erase a frame or change a program counter, cross it out rather than erase it. Use the numbers in red on the right hand side for your program counter.

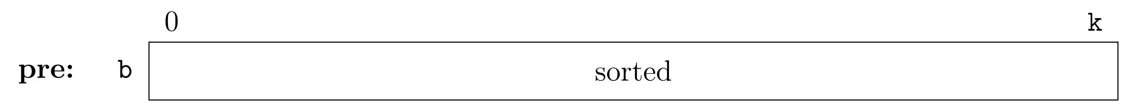


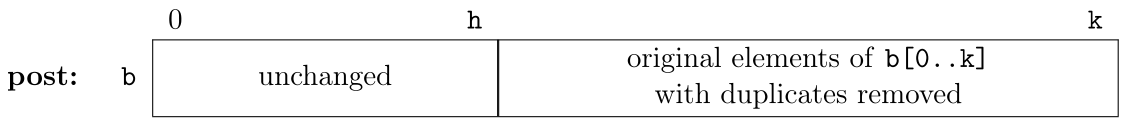
(b) [7 points] Draw the contents of global space and heap space after executing the following:

```
>>> b = [[9,6],[4,5],[7,7]]
>>> x = b[:2]
>>> x[0] = [1,2]
```



6. [12 points] **Loops and Invariants**
List segment `b[0..k]` is already sorted (in ascending order), but it may contain duplicates. We want an algorithm (using a while-loop) that will remove the duplicates, as indicated by the following pre- and post-conditions.

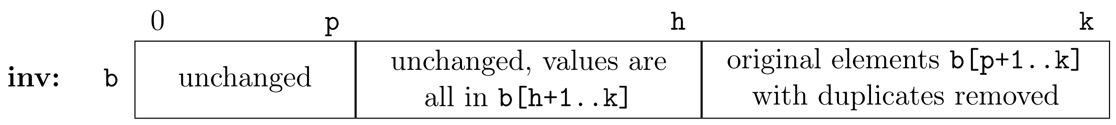




For example, for the list `b = [1, 2, 2, 2, 4, 4, 4]`, execution of the algorithm sets `h` to `k-3` (since there are 3 distinct values) and changes the contents of `b` to `[1, 2, 2, 2, 1, 2, 4]`. Note that the underlined part is the unchanged segment `b[0..h]`, and the last three values are the elements of the original array `b`, without duplicates.

Fortunately for you, we have provided the outline of the algorithm below, including the invariant. **You must use this invariant to get full credit**. In addition, keep the following in mind:

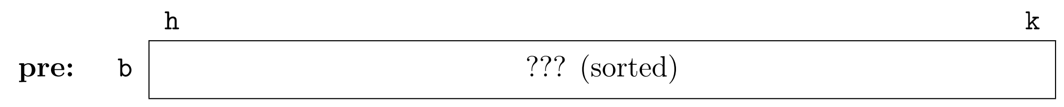
1. The invariant indicates that `b[h+1..k]` contains all the non-duplicated values in `b[p+1..k]`. So `b[p]` is the next value to check.
2. The body must determine whether `b[p]` is a duplicate (e.g. whether it already appears in `b[h+1..k]`). How can this test be done simply, using the fact that b is sorted?

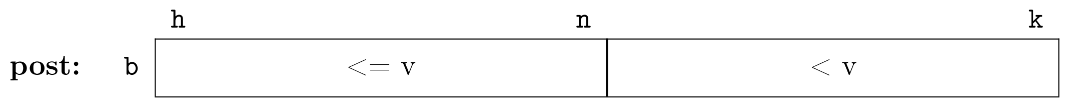


```
# Assume 0 <= k, so the list segment has at least one element
p = k-1
h = k-1
# inv: b[h+1..k] is original b[p+1..k] with no duplicates
#      b[p+1..h] is unchanged from original list w/ values in b[h+1..k]
#      b[0..p] is unchanged from original list

while 0 <= p:
    if b[p] != b[p+1]:
        b[h] = b[p]
        h = h-1

    p = p-1
```

Last Name: ________________ First: ________________ Netid: ________ Section ________

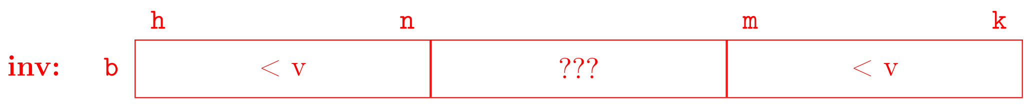
7. [16 points total] **Binary Search**
As we have seen in class, binary search allows us to search a list much faster if it is already sorted. Binary search is given a list `b` and a value `v`, not necessarily in the list. It does not modify the list, but instead finds the value `n` satisfying the post-condition below:





When done, it uses the value at position `n` to determine if `v` is in the list. Note that this version of the post-condition returns the position of the last instance of `v`.

(a) [6 points] Give the loop invariant for searching using **binary search**. You must write the invariant using our pictoral representation. Your placement of variables should be clear.



(b) [10 points] Implement the function `binary_search` below. You do not need to restate the invariant above, but you must follow it. Answers that do not follow the invariant will lose points even if correct.

```
def binary_search(v,b,h,k):
    """Return:  Last position of v in b[h..k]; -1 if v is not in b[h..k]
    Precondition: v an int, b a list of ints, h <= k are positions in b""".
    # Satisfy the invariant
    n = h-1
    m = k+1
    # The middle position of the range
    mid = (n+m)/2

    # invariant; b[h..n] <= v, b[n+1..m-1] unknown, b[m..k] > v
    while n < m-1:
        if b[mid] > v:
            m = mid
        else:  # b[mid] <= c
            n = mid
        # Compute a new middle.
        mid = (i+j)/2
    # post:  n == m-1 and b[0..n] <= c and b[m..k] > c
    return i if (i < len(b) and b[i] == c) else -1
```

8. [12 points total] **Potpourri**

(a) [2 points] What is the printed output when executing the code below?

```
>>> d = {'js1':'John','js2':'John','wmw2':'Walker'}
>>> print d['wmw2']
>>> print ('John' in d)
```

```
'Walker'
False
```

(b) [4 points] What is printed output when executing the code on the right?

```
def foo(a):
    b = True
    for k in range(len(a)):
        b = b and decr(a,k)
    return b

def decr(a,k):
    a[k] = a[k]-1
    return a[k] >= 0
```

```
>>> a = [1,2,3,4]
>>> print foo(a)
>>> print a
>>> print foo(a)
>>> print a
```

```
True
[0,1,2,3]
False
[-1,1,2,3]
```

(c) [4 points] Describe the four main steps that happen when you call a constructor.

- It creates a new object (folder) of the class, which is empty.
- It puts the folder into heap space
- It executes the initializer method `__init__` on this new folder.
- It returns the object (folder) name as final value of expression.

(d) [2 points] Name the four types of variables we have seen in class.

- Global variables
- Local variables
- Parameters
- Attributes