READ BEFORE YOU BEGIN

The CS61A staff will go over these questions during the review session. Answers and explanations will be posted on the course website on Friday.

We strongly suggest that you attempt these questions by yourself prior to the review session. However, do not be discouraged if you find these questions difficult. Rather than modeling a real exam (some easy questions, some average, some hard) we opted to put challenging problems on this review sheet. Our thinking is that by exposing you to these questions early, you will have time to digest these difficult concepts, which should improve your exam performance overall (and make those easy problems a piece of cake).
1. Lists and Recursion

Quicksort is a fast sorting algorithm that can be used to sort an array of numbers (http://en.wikipedia.org/wiki/Quicksort). The steps are:

1. Pick the first element of the list as the “pivot”
2. Divide the rest of the list to two buckets
   a. one that contains elements less than or equal to the pivot
   b. one that contains elements greater than the pivot
3. Quicksort the two buckets from (2)
4. Return [sorted bucket a] + [pivot] + [sorted bucket b].

def quicksort(lst):

2. Mutable Data, Nonlocal

def long_term_memory():
    remember = []
    def memorize(s):
        remember.append(s)
        return remember
    return memorize

def short_term_memory():
    remember = ""
    def memorize(s):
        rtn = remember
        remember = s
        return rtn
    return memorize
Given the definitions above, what does the following print? Either write out the value, or put “ERROR” and why.

```python
>>> short = short_term_memory()
>>> short("a")


>>> long = long_term_memory()
>>> long("a")


3. Mutable Data, Nonlocal

Draw the environment diagram for the following code and provide return values when prompted

```python
def make_bunny(num_carrots):
    def dispatch(msg):
        if msg == 'num_carrots':
            return num_carrots
        elif msg == 'plant':
            def plant():
                nonlocal num_carrots
                num_carrots += 1
            plant()
        elif msg == "get_fat":
            def get_fat():
                "eat all the carrots"
                num_carrots = 0
            get_fat()
        else:
            print("bunnies don’t know how to do that")
        return dispatch

>>> funny = make_bunny(10)
>>> funny('num_carrots')


>>> funny('plant')
>>> funny('num_carrots')


>>> funny('get_fat')
>>> funny('num_carrots')

```
4. Constraint Programming

So far, we've defined the adder constraint such that it could only take in two inputs,

\[ a + b = c \]

However, it'd be convenient if we had an add constraint that could accept any number of arguments, like:

\[ a + b + c + d = e \]

Implement the `n_adder` procedure that, given `n` connectors \(c_1, c_2, ..., c_n\) represents the equation:

\[ c_1 + c_2 + ... + c_{n-1} = c_n \]

Example:

```python
def sum_connectors(conns):
    sum = 0
    for conn in conns:
        if conn['has_val']():
            sum += conn['val']
    return sum

def is_rhs(conn, conns):
    return conn == conns[-1]
```

```python
>>> # Represent the constraint a + b + c = d
>>> a, b, c, d = [make_connector(name) for name in ('a', 'b', 'c', 'd')]
>>> n_adder(a, b, c, d)
>>> a['set_val']('user', 2)
a = 2
>>> b['set_val']('user', 4)
b = 4
>>> d['set_val']('user', 14)
d = 14
c = 8
```
def n_adder(*args):
    def new_value():
        "Fill me in"
    def forget_value():
        "Fill me in"

    constraint = {'new_val': new_value, 'forget': forget_value}
    for connector in args:
        connector['connect'](constraint)
    return constraint
5. Tree Traversal, Orders of Growth

A trie (from the word retrieval, pronounced “tree”) is a tree data structure that is capable of storing a dictionary of words very efficiently. It is the de facto standard for storing large language corpora in many mobile phones. (http://en.wikipedia.org/wiki/Trie)

The diagram above shows an example trie that represents a small dictionary. A trie begins with a blank node, which represents the empty string. Each node of the tree is one alphabet, and has a Boolean variable word associated with it. To check whether a word exists in our dictionary, we take the word and travel down the trie one alphabet at a time. If at any particular node the next node we need is not one of the children, then the word is not in the dictionary. If we do reach the end of the word, we look at the node for the last character, and if word == True then the word exists in the dictionary. For example, the trie above has the words ['a', 'i', 'is', 'he', 'hey'].

Class TrieNode(object):

    def __init__(self, char=None, children = {}):
        self.char = char #None for root only
        self.children = children
        self.word = False

    def add_child(self, node):
        self.children[node.char] = node

    def get_child(self, c):
        if c in self.children:
            return self.children[c]
        else:
            return None
We have provided an implementation for trie nodes above. Below is a skeleton implementation for a Trie class. Complete it!

```
Class Trie(object):

    def __init__(self):
        self.root = Node()

    def add_word(self, word):
        """Fill me in!""

    def contains(self, word):
        """Fill me in!""
```

Why is this efficient for storing a big dictionary of words? Specifically, write a function f(n) where n is the length of the word, such that $\Theta(f(n))$ is the order of growth of `contains`. 
6. Object Oriented Programming

a) Define (Fill in) the following above-the-line Worker class in the below-the-line implementation we saw in class.

class Worker(object):
    def __init__(self, money):
        self.salary = 100
        self.money = money
    def work(self):
        self.money += self.salary
    def new_salary(self, amount):
        self.salary = amount

def make_worker_class():
    def __init__(self):
        return make_class()
b) Suppose we modify the Worker class to include a shop method. Then given the following Celebrity class (which inherits from this modified Worker class), fill in the empty lines.

**Line 1:** A celebrity is just as hard of a worker as any other professional; they start out with no money and thus should be initialized in the same way as any other worker. However, celebrities are special in other ways, and thus have been modified to reflect these special features.

**Line 2:** Every time a new celebrity is made, the happiness of all of the celebrities goes down by one, because they are that less famous with someone new in the spotlight.

class Worker(object):
    def __init__(self, money):
        self.salary = 100
        self.money = money
    def work(self):
        self.money += self.salary
    def new_salary(self, amount):
        self.salary = amount
    def shop(self):
        return self.salary > 999

class Celebrity(Worker):
    happiness = 100
    def __init__(self):
        self.salary = Celebrity.happiness*1000
        self.new_salary = 42
    def splurge(self):
        if self.shop():
            print("you just bought 10,000 shoes")

Now, predict what the python interpreter will print in response to the below calls:

```python
>>> Cher = Celebrity()
>>> Cher.salary

>>> Cher.splurge()

>>> Cher.happiness

>>> Cher.new_salary

>>> Worker.new_salary(42)
```