EXTRA MIDTERM REVIEW

COMPUTER SCIENCE MENTORS 61A

March 12 to March 14, 2018

Midterm Review

This worksheet contains some extra problems beyond the weekly worksheet that may be good practice. The topics covered include:

- Lists (non-mutation)
- List Comprehensions
- Nonlocal
- Orders of Growth

This list is by no means exhaustive, and as we are not officially affiliated with the course, we cannot guarantee that these topics will show up on the midterm either. However, these are topics that historically do show up.

1. What would Python display? Draw box-and-pointer diagrams to find out.

   (a)
   ```
   L = [1, 2, 3]
   B = L
   B
   ```

   Solution: `[1, 2, 3]`

   (b)
   ```
   A = L[1:3]
   L[0] = A
   L = L + A
   B
   ```

   Solution: `[[2, 3], 2, 3]`

2. Write a list comprehension that accomplishes each of the following tasks.

   (a) Square all the elements of a given list, `lst`.

   Solution:
   ```
   [x ** 2 for x in lst]
   ```

   (b) Compute the dot product of two lists `lst1` and `lst2`. *Hint*: The dot product is defined as $\text{lst1}[0] \cdot \text{lst2}[0] + \text{lst1}[1] \cdot \text{lst2}[1] + \ldots + \text{lst1}[n] \cdot \text{lst2}[n]$. The Python **`zip`** function may be useful here.

   Solution:
   ```
   sum([x * y for x, y in zip(lst1, lst2)])
   ```

   (c) Return a list of lists such that `lol = [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]`.

   Solution:
   ```
   [[x for x in range(y)] for y in range(1, 6)]
   ```

   (d) Return the same list as above, except now excluding every instance of the number 2: `lold = [[0], [0, 1], [0, 1], [0, 1, 3], [0, 1, 3, 4]])`.

   Solution:
   ```
   [[x for x in range(y) if x != 2] for y in range(1, 6)]
   ```

3. (a) Draw the environment diagram that results from running the code.

```
def what(a, b):
    x = a
    def ha(ha):
        nonlocal x
        x = ha * 2
        return x
    return b(ha(x), x)

what(4, lambda x, y : x)
```

Solution: https://goo.gl/aGQF7c

(b) Write the simplest possible function that does the same thing as `what` for any input `a`, `b`.

Solution:

```
def foo(a, b):
    a *= 2
    return b(a, a)
```

Note: Having the body of the function as just `return b(2 * a, 2 * a)` is incorrect. To test this out, try passing in `[1, 2, 3]` for `a` and `lambda x, y: x is y` for `b`.

4. **Fast Exponentiation:** in this problem, we will examine a real-world algorithm used to improve the speed of calculating exponents.

   (a) First, express the runtime of the naive exponentiation algorithm in big-O notation.

   ```
   def exp(b, n):
       if n == 0:
           return 1
       else:
           return b * exp(b, n - 1)
   ```

   Solution: $O(n)$. n decreases by 1 each call, so there are naturally n calls.

   (b) Now, express the runtime of the fast exponentiation algorithm in big-O notation.

   ```
   def fast_exp(b, n):
       if n == 0:
           return 1
       elif n % 2 == 0: # Assume square runs in constant time
           return square(fast_exp(b, n // 2))
       else:
           return b * fast_exp(b, n - 1)
   ```

   Solution: $O(\log n)$. n is halved each call, so the number of calls is the number of times n must be halved to get to 1. This is $\log n$.

   (c) What about this slightly modified version of `fast_exp`?

   ```
   def fast_exp(b, n):
       for _ in range(50 * n):
           print("Killing time")
       if n == 0:
           return 1
       elif n % 2 == 0:
           return square(fast_exp(b, n // 2))
       else:
           return b * fast_exp(b, n - 1)
   ```

   Solution: $O(n)$. Ignore the constant term. The first call will perform n operations, the second call will perform $n/2$ operations, the third will perform $n/4$ operations, etc. Using geometric series, we see this adds up to $2n$, which is n if we ignore constant terms.

5. **Mysterious loops:** What is the order of growth in time for the following functions? Use big-O notation.

   (a)
   ```
   def mystery(n):
       for i in range(n):
           while i % 2 != 0:
               print(i)
               i = i - 1
           print("Done")
   ```

   Solution: $O(n)$. The work for when i is divisible by two is constant. Subtracting one will immediately allow us to exit the `while` loop. Therefore, we can concentrate on just the outer loop.

   (b)
   ```
   def fun(n):
       for i in range(n):
           for j in range(n * n):
               if j == 4:
                   return -1
               print("Fun!")
   ```

   Solution: $O(1)$. Inner loop always immediately exits after running for 4 iterations, independent of n.