Problem Set 1

This problem set is due **September 25** at **11:59PM**.
Solutions should be turned in through the course website in PS or PDF form using LaTeX. The course website has links to a number of editors that are useful for writing in LaTeX.
It is recommended that you download the LaTeX source for this problem set which includes place-holders for solutions.

1. Asymptotic Notation **Collaborators:** LIST COLLABORATORS HERE

   Decide whether these statements are **True** or **False**. You must briefly justify all your answers to receive full credit.

   (a) $f(n) = \Omega(g(n)) \implies g(n) = O(f(n))$

   Solution: INSERT ANSWER HERE

   (b) $f(n) = O(g(n)) \wedge f(n) = \Omega(h(n)) \implies g(n) = \Theta(h(n))$

   Solution: INSERT ANSWER HERE

   (c) $f(n) = O(g(n)) \wedge g(n) = \Omega(f(n)) \implies f(n) = \Theta(g(n))$

   Solution: INSERT ANSWER HERE

2. Binary Search **Collaborators:** LIST COLLABORATORS HERE

   In *Problem Solving With Algorithms And Data Structures Using Python* by Miller and Ranum, two examples are given of a binary search algorithm. Both functions take a sorted list of numbers, `alist`, and a query, `item`, and return true if and only if `item` $\in$ `alist`. The first version is iterative (using a loop within a single function call) and the second is recursive (calling itself with different arguments). Both versions can be found on the last page of this problem set.

   Let $n =$ `len(alist)`.

   (a) What is the runtime of the iterative version in terms of n, and why?

   Solution: INSERT ANSWER HERE

   (b) What is the runtime of the recursive version in terms of n, and why?

   Solution: INSERT ANSWER HERE

   (c) Explain how you might fix the recursive version so that it has the same asymptotic running time as the iterative version (but is still recursive).

   Solution: INSERT ANSWER HERE

3. Set Intersection **Collaborators:** LIST COLLABORATORS HERE

Python has a built in `set` data structure. A `set` is a collection of elements without repetition.

In an interactive Python session, type the following to create an empty set:

```
s = set()
```

To find out what operations are available on sets, type:

```
dir(s)
```

Some fundamental operations include `add`, `remove`, and `__contains__` and `__len__`. Note that `__contains__` and `__len__` are more commonly called with the syntax `element in set` and `len(set)`. All four of these operations run in constant time.

For this problem, we will be analyzing the runtime of `intersection`, `intersection_update`, `union`, and `update`, on two sets, s and t.

(a) What do each of those four operations do? Use the Python `help` command. Refer to http://docs.python.org/ as necessary.

Solution:

`intersection`	INSERT ANSWER HERE
`intersection_update`	INSERT ANSWER HERE
`union`	INSERT ANSWER HERE
`update`	INSERT ANSWER HERE

(b) Using Θ notation, how long do you conjecture each of the four operations will take in terms of $|s|$, $|t|$, $|s \cup t|$, and $|s \cap t|$? Give reasons.

Solution:

`intersection`	INSERT ANSWER HERE
`intersection_update`	INSERT ANSWER HERE
`union`	INSERT ANSWER HERE
`update`	INSERT ANSWER HERE

(c) Now try these operations out using a variety of values for $|s|$, $|t|$, $|s \cup t|$, and $|s \cap t|$. You may wish to use the Python modules `profile` or the more lightweight `timeit`. A good description of how to use `timeit` is available at http://www.diveintopython.org/performance_tuning/timeit.html

Describe your methods and results. Try to give a simple but reasonably accurate formula that fits your experimental results. Discuss any discrepancies between your conjectures in part (b) and your experimental results.

Solution: INSERT ANSWER HERE

Iterative Version:

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)/2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found
```

Recursive Version:

```
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)/2
        if alist[midpoint]==item:
            return True
        else:
            if item<alist[midpoint]:
                return binarySearch(alist[:midpoint],item)
            else:
                return binarySearch(alist[midpoint+1:],item)
```