Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — February 3rd, 2009
Professors Sivan Toledo and Alan Edelman — Handout 2

Problem Set 1

This problem set is divided into two parts: Part A problems are programming tasks, and Part B problems are theory questions.

Part A questions are due **Tuesday, February 17th** at **11:59PM**.

Part B questions are due **Thursday, February 19th** at **11:59PM**.

Solutions should be turned in through the course website in PDF form using LaTeX or scanned handwritten solutions.

A template for writing up solutions in LaTeX is available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A: Due Tuesday, February 17th

1. **(20 points)** Running Time

   Version 6 of our Document Distance code uses an algorithm called *merge sort* to improve upon the $\Theta(n^2)$ running time of insertion sort. (We'll talk more about merge sort and other sorting algorithms in a few weeks.)

   You can find an implementation of merge sort on page 3 of this document.

   (a) **(10 points)** Determine experimentally the running time of mergesort, by running it on different-sized lists of random numbers. (After you `import random`, you can get a random floating-point number using the `random.random()` function.)

   Fill in the following chart. Include in your PDF submission a snippet of code that determines one of the entries in the chart.

	$\|s\| = 10^2$	$\|s\| = 10^3$	$\|s\| = 10^4$	$\|s\| = 10^5$
time in ms				

   There are a number of ways to time code. You can use the `timeit` module[1] Alternatively, if you have `ipython` installed,[2] you can use the more user-friendly builtin `%timeit` command. Make sure you check what the default number of iterations for your timing command is! By default, `Timer.timeit()` in the

[1] See `http://www.diveintopython.org/performance_tuning/timeit.html` for a good description of how to use the `timeit` module.

[2] See `http://ipython.scipy.org/`. Highly recommended.

`timeit` module runs your command *one million* times. You should change it to be high enough so your results don't have too much variance, but low enough that you don't die of boredom.

(b) **(10 points)** Give an approximate formula for asymptotic running time of merge sort based on your experiments. Justify your answer by dividing your numbers from the chart above by the formula, and showing that the result is approximately constant.

2. **(30 points)** Set Intersection

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()
```

We can also create a set from a list:

```
l = [1, 2, 3]
s = set(l)
```

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

```
dir(s)
```

Some fundamental operations of sets include `add`, `remove`, and `__contains__` and `__len__`. Note that the `__contains__` and `__len__` methods are more commonly called with the syntax `element in s` and `len(s)`. (While you *can* use `s.__contains__(element)` and it will work just fine, it's not very nice-looking, and people will look at you funny if you write it that way.) All four of these operations run in **constant time** i.e. $O(1)$ time.

`s.intersection(t)` that takes two sets, s and t, and returns a new set containing all the elements that occur in both s and t. We will use `intersection` in a new version of the Document Distance code from the first two lectures.

In the Document Distance problem from the first two lectures, we compared two documents by counting the words in each, treating these counts as vectors, and computing the angle between these two vectors. For this problem, we will change the Document Distance code to use a new metric. Now, we will only care about words that show up in both documents, and we will ignore the contributions of words that only show up in one document.

Download `ps1.py`, `docdist7.py`, and `test-ps1.py` from the class website. `docdist7.py` is mostly the same as `docdist6.py` seen in class, however it does not implement `vector_angle` or `inner_product`; instead, it imports those functions from `ps1.py`. Currently, `ps1.py` contains code copied from `docdist6.py`, but you will need to modify this code to implement the new metric.

Merge sort code for Part A, Problem 1:

```
def merge_sort(A):
    """
    Sort list A into order, and return result.
    """
    n = len(A)
    if n==1:
        return A
    mid = n//2      # floor division
    L = merge_sort(A[:mid])
    R = merge_sort(A[mid:])
    return merge(L,R)

def merge(L,R):
    """
    Given two sorted sequences L and R, return their merge.
    """
    i = 0
    j = 0
    answer = []
    while i<len(L) and j<len(R):
        if L[i]<R[j]:
            answer.append(L[i])
            i += 1
        else:
            answer.append(R[j])
            j += 1
    if i<len(L):
        answer.extend(L[i:])
    if j<len(R):
        answer.extend(R[j:])
    return answer
```

(a) **(15 points)** Modify `inner_product` to take a third argument, `domain`, which will be a `set` containing the words that occur in both texts. Modify the code so that it only increases `sum` if the word is in `domain`.

(b) **(15 points)** Modify `vector_angle` so that it creates sets of the words in both `L1` and `L2`, takes their intersection, and uses that intersection when calling `inner_product`.

(Hint: You can probably make the needed changes in under a dozen lines of code.)

Run `test-ps1.py` to make sure your modified code works. The same test suite will be run when you submit `ps1.py` to the class website.

Your code should not take significantly longer to run with the new metric. (Why not?)

Submit `ps1.py` on the class website. All code submitted for this class will be checked for accuracy, asymptotic efficiency, and clarity.

Part B: Due Thursday, February 19th

1. **(25 points)** Asymptotic Growth

   For each group of six functions below, rank the functions by increasing order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_6$ of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $\ldots$, $g_5 = O(g_6)$. Partition each list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

   (a) (8 points) Group 1:

   $$\begin{array}{lll} f_1(n) = 10^{6006}n^2, & f_2(n) = 1/n, & f_3(n) = n^{6.006} \\ f_4(n) = n, & f_5(n) = 1/6.006, & f_6(n) = 6.006n \end{array}$$

   (b) (9 points) Group 2:

   $$\begin{array}{lll} f_1(n) = n \log n, & f_2(n) = \log(\log(n)), & f_3(n) = \binom{n}{50} \\ f_4(n) = n^{1/50}, & f_5(n) = \log(n), & f_6(n) = n^{50} \end{array}$$

   (c) (8 points) Group 3:

   $$\begin{array}{lll} f_1(n) = 2^n, & f_2(n) = n2^n, & f_3(n) = 2^{n+1} \\ f_4(n) = 4^n, & f_5(n) = n!, & f_6(n) = 3^{\sqrt{n}} \end{array}$$

2. **(25 points)** Binary Search

   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.

In theory, the iterative version and the recursive version should have the same time complexity. However, in the following code, they are implemented with different basic Python operations, and their complexity is actually different. To help pinpoint where the difference occurs, you may want to look up operations in the Python Cost Model on the course website.

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

(a) **(8 points)** What is the runtime of the iterative version in terms of n, and why? Be sure to state a recurrence relation and solve it.

(b) **(8 points)** What is the runtime of the recursive version in terms of n, and why? Be sure to state a recurrence relation and solve it.

(c) **(9 points)** 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).

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)
```