Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — Feb 7, 2007
Professors Srini Devadas and Erik Demaine — Handout 3

Problem Set 1

Exercises are for extra practice and should not be turned in.
Exercises:

- Exercise 2.3-6 (page 37) from CLRS.

  Solution: No. You can find the place in the array to insert in $O(\log n)$ time, but you still have to push all the elements over to make room.

- Exercise 3.1-3 (page 50) from CLRS.

  Solution: $O(n^2)$ essentially means "at most" n^2. Saying the running time is "at least at most n^2" is meaningless.

- Exercise 3.1-4 (page 50) from CLRS.

  Solution:

$$2^{n+1} = 2 * 2^n = O(2^n)$$
$$2^{2n} = 4^n \neq O(2^n)$$

---

1. **(11 points)** Asymptotic Growth

   Rank the following functions by increasing order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{11}$ of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $\ldots$, $g_{10} = O(g_{11})$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$. All the logs are in base 2.

$$\binom{n}{100}, \quad 3^n, \quad n^{100},$$
$$1/n, \quad 2^{2n}, \quad 10^{100}n,$$
$$3^{\sqrt{n}}, \quad 1/5, \quad 4^n,$$
$$n \log n, \quad \log(n!).$$

Solution: The ordering is:

$1/n$	
$1/5$	
$10^{100}n$	
$n \log n$	$\log(n!)$
$\binom{n}{100}$	n^{100}
$3^{\sqrt{n}}$	
3^n	
2^{2n}	4^n

2. **(19 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.

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

   (a) **(6 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.

   Solution:

$$T(n) = T(n/2) + \Theta(1)$$
$$T(n) = \Theta(\log n)$$

   (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.

   The trick here is that the Python slice operation `alist[midpoint:]` takes time proportional to the new list.

   Solution:

$$T(n) = T(n/2) + \Theta(n)$$
$$T(n) = \Theta(n) + \Theta(n/2) + \Theta(n/4) + \cdots$$
$$T(n) = \Theta(n)$$

   (c) **(5 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).

   Solution: Too similar solutions are acceptable here. Both solutions avoid using the slice operator.

i. Change the recursive function to take two additional arguments, `first` and `last`.

ii. Leave the original function alone, but have it call a helper function that does the actual recursion, using two additional arguments, `first` and `last`.

3. **(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()
```

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 i.e. $O(1)$ time.

For this problem, we will be analyzing the runtime of `s.intersection(t)` that takes two sets, s and t, and returns a new set with all the elements that occur in both s and t. We will then use `intersection` in a new version of the Document Distance code from the first two lectures.

(a) **(5 points)** Using Θ notation, make a conjecture for the asymptotic running time of `s.intersection(t)` in terms of the sizes of the sets: $|s|$ and $|t|$. Justify your conjecture.

HINT: Think about the fundamental operations above.

Solution: You can loop through all the elements in s, and using the `__contains__` operation, check in constant time whether the element is in t. This takes time $O(|s|)$. Also acceptable is $O(|t|)$, $O(|s| + |t|)$, or $O(\min(|s|, |t|)$.

(b) **(10 points)** Determine experimentally the running time of `s.intersection(t)`, by running it with different sized sets. Fill in the following chart. Include in your PDF submission a snippet of code that determines one of the entries in the chart.

Note: there are a number of ways to time code. You can use the `timeit` module (see `http://www.diveintopython.org/performance_tuning/timeit.html` for a good description of how to use it). Alternatively, if you have `ipython` installed (see `http://ipython.scipy.org`), you can use their builtin `timeit` command which is more user friendly.

Solution: Results should look something like the following:

time in μs	$\|s\| = 10^3$	$\|s\| = 10^4$	$\|s\| = 10^5$	$\|s\| = 10^6$
$\|t\| = 10^3$	297	232	281	224
$\|t\| = 10^4$	210	2792	2976	3046
$\|t\| = 10^5$	230	3065	25535	33906
$\|t\| = 10^6$	224	2957	33010	262384

(c) **(5 points)** Give an approximate formula for asymptotic running time of `s.intersection(t)` based on your experiments. How does this compare with your conjecture in part (a)? If the results differ from your conjecture, make a new conjecture about the algorithm used.

Solution: $\Theta(\min(|s|, |t|))$. This is faster than my conjecture when s and t are of very different sizes. First, check to see which set is smaller. Then loop through that set, checking for each element whether it's in the other.

(d) **(10 points)** In the Document Distance problem from the first two lectures, we compared two documents by counting the words in each, treating theses 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 straight from `docdist6.py`, but you will need to modify this code to implement the new metric.

- Modify `inner_product` to take a third argument, `domain`, which will be a `set` containing the words in both texts. Modify the code so that it only increases `sum` if the word is in `domain`.
  Don't forget to change the documentation string at the top.
- 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`. Again, don't forget to change the docstring at the top.

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.

Does your code take significantly longer with the new metric? Why or why not?

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

Solution: The code should not take significantly longer with the new metric. The asymptotically slowest step is still merge sort.

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)
```