Searching

Here is a very common problem: we have a list L of data and we want to find whether a particular value x is in this list. We will represent this with function search(x, L) that returns the index of x in L if there is such an index, and it returns -1 if x is not in L. If x is in L multiple times we want search to return some index of x, but not necessarily the first index.

One obvious solution is called Linear Search. We compare x to the first element of L, the second element, and so forth. If we find a match we return its index. If we get to the end of the list without finding a match we return -1:

```
def LinearSearch(X, L):
    n = len(L)
    for i in range(0, n):
        if x == L[i]:
            return i
    return -1
```

What is the largest number of comparisons this could do in searching for x in a list of size n?

A) log(n)

B) n

C) n^2

D) I don't know. Leave me alone.

How long does LinearSearch take for a list of size n? That depends on how lucky we are; we might get a match on the first comparison. Many people rely on worst-case analyses. The worst case if x is actually in L is that we only find x after n comparisons. Of course, if x is not in L we must do n comparisons before we can return -1. The worst case running time of LinearSearch is O(n) whether x is in L or not.

There is another solution to the search problem. This one requires L to be sorted. If L is sorted we can immediately go to the middle of the list; suppose this is at index mid. If x == L[mid] we return mid and we are done. If x < L[mid] we know x is in the first half of L (because L is sorted), so we can throw out all of the elements at index mid or higher; if x > L[mid] we can throw out all elements index mid or lower. This continues until we find x or L has no remaining elements.

In practice it takes too long to "throw out" elements from a list so instead we keep variables *low* and *high* that give the smallest and largest indexes of the portion of L that might contain x

Here is a recursive helper function that does all of the work:

```
def BSearch(x, L, low, high):
    if high < low:
        return -1
    else:
        mid = (low+high)//2
        if x == L[mid]:
            return mid
        elif x < L[mid]:
            return BSearch(x, L, low, mid-1 )
        else:
            return BSearch(x, L, mid+1, high )
```

The BinarySearch function itself just calls BSearch:

```
def BinarySearch(x, L):
        return BSearch(x, L, 0, len(L)-1 )
```

How do we analyze BinarySearch? Note that each time BSearch recurses it cuts in half the region of L it is considering. If L starts with n elements, how many times can we cut it in half before we get down to one element? That is the reverse of the question: how many times can we double a number that starts at 1 before it gets up to n? Either way the answer is $\log_2(n)$. That is what a logarithm is: the exponent you need to raise its base to in order to get n. So the worst-case running time of BinarySearch is $O(\log(n))$

So which is better -- LinearSearch or BinarySearch? LinearSearch is O(n), BinarySearch is O(log(n)). Suppose we have a really large list with a million (10^6) entries. LinearSearch will do up to a million comparisons. BinarySearch will do $\log_2(10^6)$, which is about 20 comparisons. There is no contest; BinarySearch is vastly better.

Of course, BinarySearch requires the list to be sorted. To do a small number of searches it would take longer to sort L and then use BinarySearch than to just use LinearSearch.