EASIER TO READ SEARCH PROCEDURE

One of the nice things about coding is that there are many ways of writing a program to do the same thing. Not only that, there are many reasons to write code. The point of presenting code in the text, slides, and lectures to precisely explain a computer science concept. Code is more precise than a lengthy wordy description. In the first few lectures, the goal is to show how to write code in Python. Now, the goal is to discuss algorithms and their complexity. When we want to discuss the running time of an algorithm, it is important to account for all the overheads. This can be difficult when programming languages provide very concise notation for complex operations. For example, in Python, newList = oldList[:] is one simple line, but it makes a copy of the whole list. If the list is long, then the copy can take a long time (O(n) in fact). On the other hand, it is much easier to read than a "for loop" that makes the overhead of the copying obvious.

The lecture on searching is concerned with the running time of various searching algorithm -- how long does it take to find an item in an ordered list. It is important to ensure that there is not any hidden overhead such as list copying. This is why the code in the lectures use indices.

This short memo points out the Python way of describing the search algorithm in a way that is clear to understand, but may have some hidden overhead. When we are not concerned with the complexity or counting the operations, then clarity and readability is more important. It is up to you to know when to use each.

An iterative "Pythonic" search procedure:

```
def search(list, element):
    for e in list:
        if e == element:
            return True
    return False
```

The recursive way:

```
def rSearch(list,element):
    if element == list[0]:
        return True
    if len(list) == 1:
        return False
    return rSearch(list[1:],element)
```

A recursive "Pythonic" binary search procedure:

```
def rBinarySearch(list,element):
    if len(list) == 1:
        return element == list[0]
    mid = len(list)/2
    if list[mid] > element:
        return rBinarySearch( list[ : mid] , element )
    if list[mid] < element:
        return rBinarySearch( list[mid : ] , element)
    return True
```

Just for fun, someone with a little extra time may want to try to write the sorting algorithms in a "Pythonic" way.