Searching

If a list is completely unsorted, a linear search is the best you can do.

However, if the list is sorted, you can use a binary search, which is as fast as you can go.

Binary Search

Look at the midpoint. If it's the number you're looking for, you're done.

If not, the number you've found is either smaller or bigger than the number you're looking for, then you know which side of the midpoint it's on, which will allow you to ignore half of the list.

You then repeat this by finding the midpoint of the new section of the list.

In Python

You have a variable called high and a variable called low, and when you want to look at a smaller section of the list, you increase low or decrease high. At first they're set to the first and last indices of the list.

Alternate Binary Search

```
def BinarySearch(item, lst):

    while lst != []:
        mid = len(lst) // 2
        if item < lst[mid]:
            lst = lst[:mid]
        elif item > lst[mid]:
            lst = lst[mid + 1 :]
        else:
            return ?

    return None
```

It's a class exercise to see what's returned from the function at the `?`. You will also have to make some additions to the program for that.

Speed Comparison

This program (when completed) will work, but it is slower than the original binary search:

Algorithm	Time Taken
Original:	0.06s
Alternative:	98.52s

According to this table, the new type of binary search is 1642 times slower, which is a huge difference.

This is because `lst = lst[:mid]` is a very slow command, as it creates the new list by stepping through the elements of the old list. This means that we end up looking at roughly all of the elements, undoing our speed advantage from the binary aspect of the search.

A Flaw in Binary Search

Given the list `[1, 2, 2, 2, 2, 3, 3]` and asked to find 2, our current algorithm would just give whichever 2 it hits first.

Class exercise: by making a *small* modification to binary search as we have it, how could you get it to find the first 2?

Linear Search & Binary Search with Recursion

Recursive Programming Recap

- See if you can look at your problem, and find within it a slightly simpler version of the same problem.
- Find a base case.

Searching Example

- What's simpler than looking for a number in a list of 9 numbers? Looking for a number in a list of 8 numbers.
- The simplest case is then the empty list, because if you're looking for a number in the empty list, the result is always that it's not there.

Linear Search (Recursive Version)

Using lst = lst[1:] would seem to make sense intuitively, when we want to cut the first element off the list, but it's very slow, because it goes through the whole list to copy it.

Instead, we use the idea of indices we used in binary search.

For this there are two options:

- Use a helper function
- Use a default parameter (much nicer)

Helper Function:

```
# Main function:

def LinearSearch(item, lst):
    return LinearSearchFrom(item, lst, 0)


# Helper function:
# Note that if the index is len(lst), it's beyond the last index, (len(lst)
- 1)

def LinearSearchFrom(item, lst, start):
    if start == len(lst):
        return None
    elif item == lst[start]:
        return start
    else:
        return LinearSearchFrom(item, lst, start + 1)
```

Default Parameter:

```
def LinearSearch(item, lst, start = 0):
    if start >= len(lst):               <--Note: see below
        return None
    elif item == lst[start]:
        return start
    else:
        return LinearSearch(item, lst, start + 1)
```

Note: it's now >= rather than == because the user can provide an index that's out of bounds. Manning prefers to leave it as ==.

Binary Search (Recursive Version)

- This is more straightforward/intuitive than making a recursive version of linear search.

Limitation on Default Parameters

Here's a rule that's not exactly correct, but is a good working rule for now:

- Default parameters in Python must be constants.

This is why in the recursive handout we have this slightly awkward workaround:

```
def BinarySearch(item, lst, lo = 0, hi = None):

if hi == None:
    hi = len(lst - 1)
…
```

Iteration vs. Recursion

For each of these methods of searching, the iterative versions and recursive versions both work and are basically equivalent. You can choose whichever you're happier with.

We will come up against examples later where iterative programming can not (or can not easily) solve certain problems, which is where recursion can be very helpful.

Recursion with Lambda

(See notes on Assignment 19 for info on lambda.)

It is possible to write recursive programs using lambda by using the Y Combinator, which can create recursion from lambdas.