Miscellaneous Algorithms

Finding the largest and smallest thing in a list

The quick way in Python to find the largest and smallest things in a list `L` is below:

```
largest = max(L)
smallest = min(L)
```

In Java, use `Collections.max` and `Collections.min`. Sometimes, it won't be possible to use a built-in function like this, if the programming language you are using doesn't have such a function or if you need a max/min of something that isn't a simple list. For situations like these, below is the basic algorithm for finding a max. It's something most people learn in a first programming class.

```
largest = L[0]
for x in L:
    if x > largest:
        largest = x
```

We've chosen to start the `largest` variable equal to the first thing in the list. In place of that we can use any value that is guaranteed to be smaller than anything in the list. The idea of the algorithm is that the `largest` variable stores the largest item found thus far, and we loop through the list comparing each item in the list to `largest`. Any time we find something better, we update `largest`. If we instead want to find the min, the only change needed is replacing the > symbol with <.

It's often useful to find where in the list that the largest item occurs. The code below can be used to do that.

```
location = 0
for i in range(len(L)):
    if L[i] > L[location]:
        location = i
```

Finding the most common element in a list

The element in a list that occurs the most often is called the *mode*. Below we'll look at a two ways to find it. These approaches also demonstrate some algorithmic approaches that are useful in other contexts.

Approach 1: We will build a dictionary called `buckets` that holds counts of how many times each item occurs. For instance, if the list is `[a,b,b,d,a,b,b]`, the dictionary will be `{a:2, b:4, d:1}`. After constructing the dictionary, we loop through to find the item in it that has the highest count. Here is the code:

```
# Build the dictionary
buckets = {}
for x in L:
    if x not in buckets:
        buckets[x] = 1
    else:
        buckets[x] += 1

# Find the item with the highest count
best_key = L[0]
for key in buckets:
    if buckets[key] > buckets[best_key]:
        best_key = key
```

The code builds the dictionary by looping through the list and adding 1 to each item's count as it does so. The first time an item is seen, its count is set to 1. Finding the item with the largest count is done using a modification of the max algorithm from the previous section.

The built-in Python `collections` library has a nice object called a `defaultdict` that can be used to shorten the first part of the code above. It would look like this:

```
buckets = defaultdict(int)
for x in L:
    buckets[x] += 1
```

The `defaultdict` class is like a dictionary except that whenever we try to access an item that is not in the dictionary, a new entry is created with a default value equal to 0. That saves us the trouble of the extra if statement in the original code. The `defaultdict` class can be used with other data types besides integers.

The `collections` library has another nice object called a `Counter` that automatically builds a dictionary-like object of counts for us. We could replace the first part of the code above with the single line below..

```
buckets = Counter(L)
```

This is a very useful object that should be in your toolbox. Also, as we'll cover in a later section, we can use a key argument with Python's `max` function that tells it how to determine the max. Using this gives a particular short way to find the mode:

```
def mode(L):
    buckets = Counter(L)
    return max(buckets, key=lambda x:buckets[x])
```

Approach 2 The approach above runs in O(n) time, which is pretty good. If the list is already sorted, we have another O(n) approach that is a little faster since it only requires looping through the list once. This approach uses O(1) space, as opposed the Approach 1 which can use O(n) space in the worst case.

The basic idea is if the list is already sorted, then equal items all end up next to each other. For instance, if the list is `[a,b,b,d,a,b,b]`, then it will be sorted into `[a,a,b,b,b,b,d]`. We can loop through the list, keeping a count, until we get to a point where the current element is different from the next element. At this point, we compare the current count to the highest count found thus far, and update that highest count if needed. Then we reset the count back to 1 and continue along in the same way. This allows us to build up counts of how many of each item there are, and we keep track of the largest count as we move through the list.

```
biggest_count = 0
count = 1
for i in range(len(L)-1):
    if L[i] != L[i+1]:
        if count > biggest_count:
            biggest_count = count
            mode = L[i]
        count = 1
    else:
        count += 1
if count > biggest_count:
    mode = L[i]
```

The reason for the additional check after the loop is in case the item that occurs most often is the last thing in the list. The code in the loop won't catch that. The code above won't work if the list is empty or has only one thing in it. We could have added a couple of simple checks to handle those cases, but we have chosen not to in order not to obscure the main idea of the algorithm.

As an example of how this works, let's look at the list `[a,a,b,b,b,b,d]` The code starts with `count=1`. The first time through the loop it compares the items at indices 1 and 2, sees they are equal, and updates `count` to 2. The next iteration, it compares the items at indices 2 and 3, and sees they are different (`a` $\neq$ `b`). At this point, we set `biggest_count` to 2 and set `mode` to `a`. We also set `count` back to 1.

For the next few iterations, we will have `L[i]` equal to `L[i+1]`. This will cause `count` to increase until it gets to 4. This is counting how many `b`'s there are in the list. Once we get to `i=5`, we have `L[i]` equal to `b` and `L[i+1]` equal to `c`, which are not equal, so we run the code in the `if` block. We compare `count`, which is 4 to `biggest_count`, which is 2, and since `count` is larger, we update `biggest_count` to 4 and we update `mode` to `b`.

We also set `count` back to 1. At this point, `i` then increases to 6, which ends the loop. The value of `count` is 1, which is a count of how many d's there are. We compare this to `biggest_count`, but `count` is smaller, so there is no need to change the `mode` variable.

Shuffling

Shuffling a list has a variety of uses. For instance, we can use it to randomize the order of turns in a game or shuffle a list of cards for a card game. Python's `random` library has a function called `shuffle` and Java has `Collections.shuffle`. Many programming languages don't have a built-in shuffle function. There are several ways to implement one. A simple approach to shuffle a list is to randomly pick two items in the list, swap them, and repeat that process a bunch of times, maybe around 3 times the size of the list.

A quicker approach is the following: Pick a random index 0 or greater and swap the item at that index with the item at index 0. Then pick a random index 1 or greater and swap the item at that index with the item at index 1. Then pick a random index 2 or greater and swap the item at that index with the item at index 2. Keep going with this process until you reach the end of the array. This is an O(n) approach. Here is the code:

```
for i in range(len(L)-1):
    rand_index = randint(i, len(L)-1)
    L[i], L[rand_index] = L[rand_index], L[i]
```

This code makes use of a Python shortcut for swapping two variables. If we want to swap variables `a` and `b`, use `a,b = b,a`. In most programming languages, to swap things we would need to introduce a third variable.