Searching

We'll look at several ways of searching through a list to see if it contains a specific item.

Linear search

The simplest approach to searching is to loop through the list and compare each item of the list to see if it's the item we want. This is called a *linear search* and it runs in O(n) time. Here is the code:

```
result = False
for x in L:
    if x == item:
        result = True
        break
```

The `break` isn't strictly required, but it can save time. Once the item is found, there is no need to do any further checking. Here is the linear search coded as a function:

```
def search:
    for x in L:
        if x == item:
            return True
    return False
```

A common mistake people make when coding this is to use an if/else. The problem with doing this is be that if the first item is not a match, the function would return `False` without ever checking anything else in the list. Below is a variation on the linear search that returns the location of the item and -1 if it is not in the list:

```
result = -1
for i in range(len(L)):
    if L[i] == item:
        result = i
        break
```

Python's `in` operator when used on a list uses a linear search, as does Java's `contains` method for ArrayLists.

Binary search

If the list is already sorted, there is a much faster algorithm, called *binary search*. Let's look at it with an example. Suppose we're looking for the item `w` in the list below.

```
[a,b,b,c,e,f,h,h,h,k,m,o,p,p,q,q,q,s,t,u,w,y,z,z]
```

We start by looking at the middle item. There are 24 items in the list, and the middle item is the 12th one, which is `o`. Comparing `w` to `o`, we see `w` is greater. Since the list is in order, this tells us that if `w` is in the list, it must be in the second half of the list, between `o` and `z`. We can throw out the entire first half of the list from our search. We next repeat the process on the second half of the list. We look at the middle item of the second half of the list. This turns out to be `s`, and we compare it to `w`. We see that `w` is greater, so that means that `w` must be in the second half of this part of the list. This allows us to throw out the first half of this part of the list, which is 1/4 of the original list. We repeat the process again on the portion of the list from `t` to `z`. The midpoint of this is `w`, which is the item we are looking for, so the search ends.

Binary search works this way in general. At each step we are look at a portion of the list. We find the middle element of that portion and compare that to the element being searched for. If they are equal we are done. Otherwise, we throw out half of the portion being searched and repeat the search on the other half. This continuous throwing away of half of the list means that we very quickly zero in on the location of the item we're search for.

For instance, if we have 1024 items, after the first step we will be down to 512 items to search. After the second step, it will be 256. It will continue through 128, 64, 32, 16, 8, 4, 2, 1, meaning that we need at most 10 steps to find the item or determine it is not there. Upping the list size to 1,000,000 items, only increases the number of steps to 20. A list size of 1 billion corresponds to a max of 30 steps. This is a logarithmic algorithm, O($\log n$). Even with a huge number of items, the number of steps stays pretty small. Here is the code for the algorithm:

```
def binary_search(L, item):
    lo = 0
    hi = len(L) - 1
    while hi >= lo:
        mid = (hi + lo) // 2
        if L[mid] == item:
            return True
        elif L[mid] < item:
            lo = mid + 1
        else:
            hi = mid - 1
    return False
```

We use variables called `lo` and `hi` to keep track of which portion of the list we are searching, with `lo` starting out at the first element and `hi` at the last. We use `mid=(hi+lo)//2` to find the index of the item in the middle of the portion of the list being searched. The `//` operator rounds down in case the division doesn't come out evenly. In Java, you could just use `mid=(hi+lo)/2`.[1] The way we throw out one half of the list and just look at the other half is by changing the `lo` or `hi` variables based on the result of the comparison. If we get to a point where the `lo` and `hi` variables equal or even cross over each other, that means the item being searched for is not in the list. Let's look at how the `lo`, `mid`, and `hi` variables evolve as we search the list given earlier for the item `u`. At the start, `lo=0` and `hi=23`. We compute `mid` via `(23+0)//2`, which is 11.

```
0                   11                  23
a b b c e f h h h k m o p p q q q s t u w y z z
^                   ^                   ^
lo                 mid                  hi
```

Since `t` is greater than `o`, we set `lo` equal to `mid+1`, which means we now have `lo=12` and `hi=23`. We recompute `mid` by doing `(23+12)//12`, which is 17.

```
                      12        17          23
a b b c e f h h h k m o p p q q q s t u w y z z
                      ^         ^           ^
                      lo       mid          hi
```

Since `t` is greater than `s`, we set `lo` equal to `mid+1`, which means `lo=18` and `hi=23`. The new value of `mid` is 20

```
                                 18  20    23
a b b c e f h h h k m o p p q q q s t u w y z z
                                 ^   ^     ^
                                 lo mid    hi
```

Since `t` is less than `u`, we set `hi` equal to `mid-1`, so `lo=18` and `hi=19`. We recompute `mid`, which comes out to 18. At this point, the value at `mid` is `t`, which is what we're looking for, and the search ends.

```
                                   18 19
a b b c e f h h h k m o p p q q q s t u w y z z
                                   ^ ^
                               mid=lo hi
```

[1] 1 A safer approach is to use lo+((hi-lo)/2) which avoids a potential integer overflow problem in (hi+lo)/2. This is important in Java, but not Python, where integers can't overflow.

If you're implementing this yourself, there are some small details that are easy to mess up. For instance, the loop condition needs to be `hi >= lo` and not `hi > lo`. Using > instead of >= will lead to a wrong result in certain cases. Also, it's important to use `lo=mid+1` and not `lo=mid`, as there are certain lists that will cause us to get stuck in an infinite loop if using the latter.

Both Python's `in` operator and Java's `contains` method use a linear search since they can't assume that the given list is sorted. If you want to use a built-in binary search in Java, `java.util.Collections`, has one. Python doesn't have one, but the `bisect` library does have code that can help with implementing it.

Hashing

Linear search is an $O(n)$ algorithm and binary search is $O(\log n)$. We can actually get an $O(1)$ search using hashing. A *hash function* is a function that always returns an output in a restricted range. A simple example is the function $h(x) = x \bmod 100$ that takes any integer as input and returns an integer limited to the range from 0 to 99. In particular, it returns the last two digits of the integer. For instance, $h(1453) = 53$.

We can use hash functions as the basis of a data structure that can store things efficiently, specifically, having $O(1)$ running time to determine whether or not an item is in the collection, as long as the hash function is pretty good and there are not too many things stored in the collection. If the hash function is poor or if the collection is too large, then performance can degenerate to $O(n)$.

The basic idea is we store the data values using a list of lists, one list for each possible output value of the hash function, with that list storing all the data values that hash to that value. For instance, if we use $h(x) = x \bmod 100$ as our hash function, and we want to store 133, 977, 233, 114, 109, 1209, 9, and 46, we would end up with 100 lists, corresponding to the hash values 0 to 99. Most of those would end up being empty lists. Below are all the nonempty lists:

hash	list
9	[109, 1209, 9]
14	[114]
33	[133, 233]
46	[46]
77	[977]

When adding a new value, we compute its hash and then add the value to the list at the location given by that hash. To check to see if a value is there or not, we compute its hash and search the list at that hash for the value. If the hash function used is any good, then it will spread all the values evenly around the hashes, meaning no list will have more than a few things in it, giving us O(1) searching time.

The approach described above, where we use lists of lists, is called the *chaining* approach. There is another approach called *probing* that you might run into, but we won't cover it here. The hash function used as an example above isn't a particularly good one. A better one is $h(x) = ax \bmod b$, where a and b are relatively large primes. For storing strings, floats, and other data types, there are other good hash functions out there.

Python's `set` data type uses hashing, as does Java's `HashSet`. Python's dictionaries and Java's `HashMap` use hashing to store the keys.

Comparing the searches

Linear search is simple to code and understand, and it always works. But it's also the slowest of the searches. Binary search is considerably faster if the list is already sorted. For instance, in a list of one billion items, a linear search will need to check all one billion items in the worst case, while binary search will only need to check 30 items. However, if you're just searching through a list once, and the list isn't sorted, then binary search will be slower than a linear search since we would first have to sort the array, which takes at minimum $O(n)$ time. If you plan on doing multiple searches, then it is worth spending the time to sort the list in order to use the binary search.

As long as the hash function used is good and the collection is not overly large, the hash-based approach is usually the fastest. One drawback of the hashing approach is that the hash function will usually scramble the order that items are added to the set, which may not be desirable. However, Java has a class called `LinkedHashSet` that preserves the order.

Let's look at a comparison of the three approaches for the problem of finding all the words in a wordlist that are also words in reverse. This includes not only palindromes, but also a word such as `rats` that becomes the real word `star` when reversed. Here is Python code that does this and times it, given a list of words.

```
def timer(words):
    start = time.time()
    L = [w for w in words if w[::-1] in words]
    print(time.time() - start)
```

Be sure to import the `time` library. Try first passing it a list and then a set, like the ones created below.

```
word_list = [line.strip() for line in open('wordlist.txt')]
word_set = set(line.strip() for line in open('wordlist.txt'))
```

You'll need a wordlist file in order to use this. There are many that can be found online. To use binary search, replace the `if w[::-1] in words` part of the code with `if binary_search(words, w[::-1])`. When I tested it on my laptop with a list of about 78,000 words, the set approach took around .03 to .04 seconds, the binary search approach took around .3 to .4 seconds, and the list approach took over a minute. In this example, it was about a 10 times speedup to use hashing over our hand-coded binary search, and it's about a 2000 times speedup to use hashing over a linear search.