Searching (and Finding)

Suppose I have an array of length n, and I want to see if a particular number is in that array. There might be several reasons for this:

- The array is a set, and we just want to see if this number is in the set of numbers
- Maybe the array contains pointers to objects, and each object has an id, and we're searching for the object with that id. Then, we get a whole lot of information, not just whether the id is there.
- Maybe we're not checking for something equal to k, maybe we're checking if some function f(i) is true.

Let's simplify this problem to the former:

Given an array of integers (length n) and an integer k, find the first index of the array that contains the number k. If k is not in the array, return n - the size of the whole array.

Compare this to other questions we might ask:

- Count the occurrences of k.
- Put all occurrences of k in a linked list
- Find the LAST Index of k

In all of those cases, we can't stop early, so we know we'll have an O(n) loop. But in the indexAt problem we can stop early:

```
indexAt(arr, k):
  for i = 0, i < n, i++:
    if (arr[i] == k) return i;
  return i;
```

Want to see a cool optimization?

// suppose arr has some buffer space at the end of the array, not part of the array

```
indexAt(arr, k):
  set arr[n] = k
  i = 0
  while(true):
    if (arr[i] == k) return i;
    i++;
```

Removed the i < size check from the loop. Maybe it runs 50% faster! Could be done with linked lists too.

In any case, same big-Oh.
Worst Case: O(n)
Best Case: O(1)

Average Case: Always an interesting question about the average case. That's probably why we focus on the worst case. But what can be said about average?

Now we have to talk about probabilities, and how we expect the array to look. We can't answer this question in a vacuum.

So - a reasonable assumption:

- If k exists in the array, it exists once, and it is equally likely to be at any spot in the array.
- Go over formula for average spot: (1 + 2 + 3 + 4 + 5 + .. + n) / n = O(n^2) / O(n) = O(n)
- Another way of thinking about it (totally legit): maybe the random spot will be about halfway through the array, so n/2, or also O(n).

But there's another assumption. What if these are coin flips, and we're looking for the first head? And this is a fair coin. If I have an array of bits of length 1 billion - but it's random-looking bits, And I'm looking for the first 1, I don't think I would expect us to loop through all billion spots.

Formula (using probability theory): if each item has a probability p of reaching our criteria (being equal to k in this case), then on average we need to look through (1/p) spots.

- This is using probability theory, outside the scope of this course

So.. O(1/p)... or maybe O(min(1/p, n))
O(n/(np + 1))

A sorted list you can do a lot better. You can use binary search.

Binary Search algorithm:

```
l = 0, h = n - 1
while(l < h):
  m = (l + h) / 2
  if arr[m] < k: l = m + 1;
  else h = m
return l == h && arr[l] = k   // Mentioned in class, not sure if l == h is needed
```

IS THAT LAST low == high CHECK OPTIONAL ??

NOTE - check for equality is not in the loop, and even if we put it there we'd put it at the end

- Principle: check for the more likely side first

Average/Worst case scenario O(log(n)).

Silicon Valley:
https://www.youtube.com/watch?v=ig-dtw8Um_k

A point: sometimes linear search is not that bad compared to binary search!

- Algorithms that win at Big-Oh running times sometimes lose for small n, simplicity.