Binary Search, Reading

Binary search applies to sorted arrays, taking advantage of the fact that it's sorted, in a far more efficient way than the linear search of a sorted array. The approach is to choose a value in the middle of the array. If that's a match, we really got lucky! But more often, it's either higher or lower than the value we're looking for. If it's higher, then we can limit the search to the first half of the array, and if it's lower, we take the last half. The process repeats until by halving and halving we end up with one value that either matches or it doesn't.

```
int s = 0; // start of a range of the array to search
int e = n; // end of a range of the array to search
int m = 0; // the matching value's index
while (m != n) // "n" means no match was found
{
   m = (s + e - 1) / 2; // the middle element
   if (matchThis == a[m]) break; // got lucky -- matches middle element
   else if (s >= e - 1) m = n; // 1-element array and it did not match
   else if (matchThis < a[m]) e = m; // look between s and m-1
   else s = m + 1; // look between m+1 and e-1
}
```

This does not work so well with linked lists, because finding a value at a random location required traversal from the head node. But there are other linked structures that *do* lend themselves well to binary search, which we study in the next module on "trees".

Binary Search Big Oh

The big oh for this process is something we've seen before in the module on priority queues. Doubling n increases by one the number of times we have to halve the array. Delta-f is 1, and delta-n is n. That led to **O(log n)** before, and we have the same thing with binary search.

```
O(log n)
```

What About Unsorted Arrays?

For unsorted arrays, we're stuck with O(n). But here's a idea to improve upon that -- how about if we sort the array first, so we can then use O(log n) binary searching? A similar idea lead to the Shell sort. Something to think about -- is that a good idea from a big oh point of view? Yes, there will be a quiz question on this one!