Searching

In computer science, a **search data structure** is any data structure that allows the efficient retrieval of specific items from a set of items, such as a specific record from adatabase.

The simplest, most general, and least efficient search structure is merely an unordered sequential list of all the items. Locating the desired item in such a list, by the linear searchmethod, inevitably requires a number of operations proportional to the number *n* of items, in the worst case as well as in the average case. Useful search data structures allow faster retrieval; however, they are limited to queries of some specific kind. Moreover, since the cost of building such structures is at least proportional to *n*, they only pay off if several queries are to be performed on the same database (or on a database that changes little between queries).

Static search structures are designed for answering many queries on a fixed database; **dynamic** structures also allow insertion, deletion, or modification of items between successive queries. In the dynamic case, one must also consider the cost of fixing the search structure to account for the changes in the database.

Linear Search

A linear search is the most basic of search algorithm you can have. A linear search sequentially moves through your collection (or data structure) looking for a matching value.

Implementation

```
function findIndex(values, target) {

  for(var i = 0; i < values.length; ++i){

    if (values[i] == target) { return i; }

  }

  return -1;

}
```

```
findIndex([7, 3, 6, 1, 0], 6)
```

Example
Click *step* to step through the above implementation and find 6 within the following list

7
3
6
1
0

step

Characteristics
The worst case performance scenario for a linear search is that it needs to loop through the entire collection; either because the item is the last one, or because the item isn't found. In other words, if you have *N* items in your collection, the worst case scenario to find an item is *N* iterations. This is known as *O(N)* using the Big O Notation. The speed of search grows linearly with the number of items within your collection.

Linear searches don't require the collection to be sorted.

In some cases, you'll know ahead of time that some items will be disproportionally searched for. In such situations, frequently requested items can be kept at the start of the collection. This can result in exceptional performance, regardless of size, for these frequently requested items.

In The Real World
Despite its less than stellar performance, linear searching is extremely common. Many of the built-in methods that you are familiar with, like ruby's `find_index`, or much of jQuery, rely on linear searches. When you are dealing with a relatively small set of data, it's often good enough (and for really small unordered data is can even be faster than alternatives).

Beyond this though, the general concept of sequential/linear access is something that is often overlooked. The more abstract libraries get, the more risk you run of unknowingly doing something linearly. .NET's LINQ is a great example. Most of LINQ works against `IEnumerable` which only exposes a forward moving enumerator. So what do you think happens when you call the `Count()` method? Thankfully, LINQ is smart and, if possible, it'll rely on a fast `Count` or`Length` property. However, if the actual implementation doesn't have those, it'll loop through the enumerator.

That doesn't make LINQ's `Any`, or Ruby's `include?` "evil". It's just good to know what these high level methods might be doing (and often, what they are doing, is a linear search).

Binary Search

Binary search is one of the fundamental algorithms in computer science. In order to explore it, we'll first build up a theoretical backbone, then use that to implement the algorithm properly and avoid those nasty off-by-one errors everyone's been talking about.

Finding a value in a sorted sequence

In its simplest form, binary search is used to quickly find a value in a sorted sequence (consider a sequence an ordinary array for now). We'll call the sought value the *target*value for clarity. Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located. This is called the *search space*. The search space is initially the entire sequence. At each step, the algorithm compares the median value in the search space to the target value. Based on the comparison and because the sequence is sorted, it can then eliminate half of the search space. By doing this repeatedly, it will eventually be left with a search space consisting of a single element, the target value.

For example, consider the following sequence of integers sorted in ascending order and say we are looking for the number 55:

0	5	13	19	22	41	55	68	72	81	98

We are interested in the location of the target value in the sequence so we will represent the search space as indices into the sequence. Initially, the search space contains indices 1 through 11. Since the search space is really an interval, it suffices to store just two numbers, the low and high indices. As described above, we now choose the median value, which is the value at index 6 (the midpoint between 1 and 11): this value is 41 and it is smaller than the target value. From this we conclude not only that the element at index 6 is not the target value, but also that no element at indices between 1 and 5 can be the target value, because all elements at these indices are smaller than 41, which is smaller than the target value. This brings the search space down to indices 7 through 11:

55	68	72	81	98

Proceeding in a similar fashion, we chop off the second half of the search space and are left with:

55	68

Depending on how we choose the median of an even number of elements we will either find 55 in the next step or chop off 68 to get a search space of only one element. Either way, we conclude that the index where the target value is located is 7.