Linear Search

In C Programming, we looked at the problem of finding a specified value in an array. The basic strategy was:

Look at each value in the array and compare it to what we're looking for. If we see the value at any time, return that we've found it. Otherwise, if after we're done at looking through each item in the array, if we still haven't found it, then return that the value isn't in the array. In code, we have something like this:

```
int search(int array[], int len, int value) {

  int i;
  for (i=0; i<len; i++) {
    if (array[i] == value)
      return 1;
  }

  return 0;
}
```

Clearly, for an unsorted array, this algorithm is optimal. There's no way you can definitively say that a value isn't in the array unless you look at every single spot. (Similarly, there's no way you can say that you DON'T have some piece of paper or form unless you look through ALL of your pieces of paper.)

But, we might ask the question, could we find an item in an array faster if it were already sorted?

Binary Search

Consider the following game you most likely played when you were a child:

I have a secret number in between 1 and 100. Make a guess and I'll tell you whether your guess is too high or too low. Then you guess again. The process continues until you guess the correct number and your job is to minimize the number of guesses you make.

Typically, most people's first guess is 50. Here's why:

No matter whether the response is "too high" or "too low", the most number of possible values for your remaining search (either from 1-49 or 51-100) is 50.

Any other first guess and there's a possibility that the number of possible remaining values is greater than 50. (For example, if you guessed 75 and the response was "too high", then your number could be any number from 1-74, or one of 74 possibilities.)

So, the basic idea behind the game is: Always guess the number that is halfway between the lowest possible value in your search range and the highest possible value in your search range.

Now, how can we adapt this idea to work for searching for a given value in an array?

If I am given the array:

index	0	1	2	3	4	5	6	7	8
value	2	6	19	27	33	37	38	41	118

and am searching for 19, we might ask ourselves, where is "halfway in between? One guess would be to look at 2 and 118 and take their average (60). But, 60 isn't in the list and if we look at the number closest to 60, we find that it's almost at the end of the array.

Very quickly, we realize that if we are to adapt the guessing game strategy to searching in an array, that we want to search in the middle INDEX of the array. In this case, the lowest index is 0, the highest index is 8, so the middle index must be 4.

Thus, we would ask the question, "Is the number I am searching for, 19, greater than or less than the number stored in index 4, 33?

The answer is "less than", so we can modify our search range to in between index 0 and index 3. (Note that index 4 is no longer in the search space.)

From there we'd continue in this process, the second index we'd look at is index 1, since (0+3)/2 = 1. Then we'd finally get to index 2, since (2+3)/2 = 2, and find 19 in the array.

Now let's put this idea together and code it up:

```
int binsearch(int a[], int len, int value) {

  int low = 0, high = len-1;

  while (low <= high) {

    int mid = (low+high)/2;

    if (value < a[mid])
      high = mid-1;
    else if (value > a[mid])
      low = mid+1;
    else
      return 1;
  }

  return 0;
}
```

At the end of each array iteration, all we do is update either low or high. Doing so modifies our search region to be smaller than it previously was, based on the last comparison we made.

Efficiency of Binary Search

Now, let's analyze how many comparisons (guesses) are necessary when running this algorithm on an array of n items.

First, let's try n = 100:

After 1 guess, we have 50 items left,
After 2 guesses, we have 25 items left,
After 3 guesses, we have 12 items left,
After 4 guesses, we have 6 items left,
After 5 guesses, we have 3 items left,
After 6 guesses, we have 1 item left
After 7 guesses, we have 0 items left.

The reason we have to list that last iteration is because the number of items left represent the number of other possible values to search. We need to reduce this to 0. Also, note that when n is odd, such as when n=25, when we search the middle element, element #13, there are 12 elements smaller than it and 12 elements larger than it, so that's why the number of items is slightly less than 1/2 in those cases.

In the general case, we get something like:

After 1 guess, we have n/2 items left,
After 2 guesses, we have n/4 items left,
After 3 guesses, we have n/8 items left,
...
After k guesses, we have $n/2^k$ items left.

If we can find the value that makes this fraction 1, then we know that in one more guess we'll narrow down the item:

$$\frac{n}{2^k} = 1$$
$$n = 2^k$$
$$k = \log_2 n$$

This means that a binary search roughly takes $\log_2 n$ comparisons when searching for a value in a sorted array of n items. This is much, much faster than searching linearly. Consider the following chart:

n	log n
8	3
1024	10
65536	16
1048576	20
33554432	25
1073741824	30

Basically, any algorithm that takes $\log_2 n$ steps is super fast.