Showing binary search correct using strong indu

Strong induction

Strong (or *course-of-values*) induction is an easier proof technique than ordinary induction because you get to make a stronger assumption in the inductive step. In that step, you are to prove that the proposition holds for k+1 assuming that that it holds for all numbers from 0 up to k. This stronger assumption is especially useful for showing that many recursive algorithms work.

The recipe for strong induction is as follows:

1. State the proposition P(n) that you are trying to prove to be true for all n.
2. *Base case:* Prove that the proposition holds for n = 0, i.e., prove that P(0) is true.
3. *Inductive step:* Assuming the induction hypothesis that P(n) holds for all n between 0 and k, prove that P(k+1) is true.
4. Conclude by strong induction that P(n) holds for all $n \geq 0$.

Example: Binary Search

For example, consider a **binary search** algorithm that searches efficiently for an element contained in a sorted array. We might implement this algorithm recursively as follows:

```
/** Return an index of x in a.
 *  Requires: a is sorted in ascending order, and x is found in the array a
 *  somewhere between indices left and right.
 */
int binsrch(int x, int[] a, int left, int right) {
    int m = (left+right)/2;
    if (x == a[m]) return m;
    if (x < a[m])
        return find(x, a, l, m-1)
    else
        return find(x, a, m+1, r);
}
```

Because this code is tail-recursive, we can also transform it into iterative code straightforwardly:

```
int binsrch(int x, int[] a, int left, int right) {
  while (true) {
    int m = (left+right)/2;
    if (x == a[m]) return m;
    if (x < a[m])
        r = m-1;
    else
        l = m+1;
  }
}
```

Binary search is efficient and easy to understand, but it is also famously easy to implement incorrectly. So it is a good example of code for which we want to think carefully about whether it works. Just testing it may well miss cases in which it does not work correctly.

We can prove either piece of code correct by induction, but it is arguably simpler to think about the recursive

version. The problem with convincing ourselves that **`binsrch`** works is that it uses itself, so the argument becomes circular if we're not careful. The key observation is that **`binsrch`** works in a **divide-and-conquer** fashion, calling itself only on arguments that are "smaller" in some way. In what way do the arguments become smaller? The difference between the parameters **`right`** and **`left`** becomes smaller in the recursive call. This is then the variable we should choose to construct our inductive argument. Now we can follow the strong induction recipe.

1. Let P(n) be the assertion that **`binsrch`** works correctly for inputs where right−left = n. If we can prove that P(n) is true for all n, then we know that **`binsrch`** works on all possible arguments.

2. *Base Case*. In the case where n=0, we know left=right=m. Since we assumed that the function would only be called when x is found between left and right, it must be the case that x = a[m], and therefore the function will return m, an index of x in array a.

3. *Inductive Step*. We assume that **`binsrch`** works as long as left−right ≤ k. Our goal is to prove that it works on an input where left−right = k + 1. There are three cases, where x = a[m], where x < a[m] and where x > a[m].

    - Case x = a[m]. Clearly the function works correctly.
    - Case x < a[m]. We know because the array is sorted that x must be found between a[left] and a[m−-1]. So if the recursive call works correctly, this call will too. The n for the recursive call is n = m − 1 − left = ⌊(left+right)/2⌋ − 1 − left. (Note that ⌊x⌋ is the **floor** of x, which rounds it down toward negative infinity.) If left+right is odd, then n = (left + right − 1)/2 − 1 − left = (right − left)/2 − 1, which is definitely smaller than right−left. If left+right is even then n = (left + right)/2 − 1 − left = (right−left)/2, which is also smaller than k + 1 = right−left because right−left = k + 1 > 0. So the recursive call must be to a range of a that is between 0 and k cells, and must be correct by our induction hypothesis.
    - Case x > a[m]. This is more or less symmetrical to the previous case. We need to show that r − (m + 1) ≤ right − left. We have r − (m + 1) − l = right − ⌊(left + right)/2⌋ − 1. If right+left is even, this is (right−left)/2 − 1, which is less than right−left. If right+left is odd, this is right− (left + right − 1)/2 − 1 = (right−left)/2 − 1/2, which is also less than right−left. Therefore, the recursive call is to a smaller range of the array and can be assumed to work correctly by the induction hypothesis.
4. Because in all cases the inductive step works, we can conclude that **`binsrch`** (and its iterative variant) are correct!

Notice that if we had made a mistake coding the x > a[m] case, and passed **`m`** as **`left`** instead of **`m+1`** (easy to do!), the proof we just constructed would have failed in that case. And in fact, the algorithm could go into an infinite loop when right = left + 1. This shows the value of careful inductive reasoning.