Searching an array

We *develop* two linear search algorithms to search an array for a value. These algorithms are not difficult to write, and you probably wrote something like them in your first programming course. The emphasis here is on *developing* them: start with the specification, decide a loop is needed, develop the loop invariant, and finally write the loop using the four loopy questions.

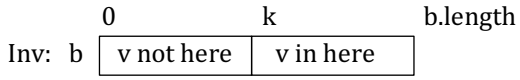
After reading this, practice developing the algorithms yourself. On a blank piece of paper, write the specification (pre- and post-conditions), then develop a loop invariant, and finally write the loop using the four loopy questions. Compare what you developed to our solution, note any differences, and ask yourself why there were differences — some differences are OK, others are just errors. This kind of practice will help you internalize the development method and later help you develop other loopy algorithms using the methodology. The methodology will begin to make more sense.

Linear search 1

Given is an array `b`, and it is known that value `v` is in it (see precondition below). Store in `k` the index of the first occurrence of `v` in `b` (see the postcondition below):

We will use a loop, which will probably start with `k` = 0 and increase `k` until `v` is found. Want the smallest of something? Start with the smallest. Want the largest? Start with the largest.

The loop invariant, based on the pre- and post-conditions, is the following:



The initialization is: `k`= 0; so that segment `b`[0..`k`-1] is empty. The loop can stop when `b`[`k`] = `v`, which is guaranteed to happen. In the loop body, making progress toward termination in the case that `b`[`k`] ≠ `v` will be to increment `k`. It is easy to see that incrementing `k` when `b`[`k`] ≠ `v` keeps the invariant true, so nothing further need be done. Therefore, the loop (with initialization) is:

```
k= 0;
// invariant: v is not in b[0..k-1] AND v is in b[k..]
while (b[k] != v)  k= k + 1;
```

Using math notation instead of array diagrams

We could have written the pre- and post-conditions like this:

Pre: `v` ∈ `b`[0..] Post: `v` ∉ `b`[0..`k`-1] and `v` = `b`[`k`]

The loop invariant could then be written as:

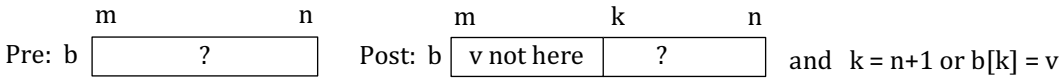
Inv: `v` ∉ `b`[0..`k`-1] and `v` ∈ `b`[0..]

Use whichever form of the assertions is easiest for you to understand.

Linear search 2

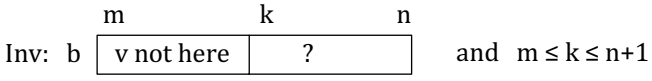
We make two modifications, partially to make sure you don't just memorize code, which almost never works, but develop starting with a specification. First, only segment b[m..n] of the array, and not the whole array, will be searched. Second, it is not guaranteed that v is in segment b[m..n]. If v is not in b[m..n], terminate with k = n+1.

Here are the pre- and post-conditions:



We could have written the postcondition as two diagrams, one showing v in b[k] and one showing b[m..n] not containing v and with k = n+1. It's easiest to write it as shown.

Again, we think of a loop that starts with k = 0 and continues to increment it. Look at the postcondition. The diagram shows what will always be true, and k has to be incremented until k = n+1 or b[k] = v is true. Therefore, we find the invariant by *deleting that last term from the postcondition*:



We did add the condition m ≤ k ≤ n+1 to make k's range explicit. It is often left out, being implicit from the array diagram, but it is best to put it in.

The invariant is initially truthified by the assignment k= m; . The loop stops when either k = n+1 or b[k] = v is true, so it must continue while is negation, k ≠ n+1 and b[k] ≠ v is true. In the repetend, incrementing k makes progress toward termination, and it is done when k ≤ n and b[k] ≠ v, so incrementing k keeps the invariant true. We end up with this algorithm:

```
k= 0;
// invariant: v is not in b[0..k-1] AND m ≤ k ≤ n+1
while (k != n+1  &&  b[k] != v)  k= k + 1;
```

This algorithm is similar to the first one. The only difference is the extra term k != n+1 in the loop condition.

Searching for the last occurrence of v

Searching for the last occurrence of v is best done by starting with k as high as possible and decreasing it until v is found or the beginning of the array (or array segment) is reached. We encourage you to redo the two algorithms developed above to find the last instead of the first occurrence of v. First, write good specifications.