We develop an algorithm (from its specification) to look for a value in a sorted array. It’s called *binary* search because at each iteration of its loop, it cuts the segment of the array still to be searched in half. This is much like a dictionary search. When you seearch a dictionary, you don’t start at the beginning and work forward. You somehow look in the middle and “throw half of it away”, depending on whether the word you are looking for is smaller or greater than the one you see.

On the second page, you will see something very strange; with a suitable specification, the algorithm works even if the array is not sorted! It’s weird but true, one of those “fake news” things that is not so fake (Jan 2019).

**The specification**

Given is a sorted array b. We won’t state again that is sorted, since the array will not be changed. The precon­dition is given below. It doesn’t say anything about the values in b —but remember that b is sorted.

The purpose of the algorithm is to store a value in k to make the postcondition (below) true.

?

0 b.length

Pre: b

0 k b.length

Post: b

> v

≤ v

Examples: Consider the array b shown to the right, and suppose v is 5. Then, k is the index of the rightmost 5 in b, as shown. On the other hand, suppose v is 7. Then k is again the index of the rightmost 5, since everything in b[0..k] <= 7 and everything in b[k+1..] > 7. In this case, one could say that v belongs after b[k].

0 k b.length

b

7 7 8 9 9

2 3 5 5 5

We can write the postcondition as b[0..k] ≤ v < b[k..]. What if b.length is 0, meaning that the array is empty? Yes,in Java, one can create an array with 0 elements. In that case, since v is not in the array, setting k to -1 satisfies the postcondition, for then the postcondition reduces to b[0..-1] ≤ v < b[0..-1] —remember: b.length = 0 in this case.

**The loop invariant**

We combine the precondition and postcondition to form the invariant shown below. We have introduced a variable t to mark the boundary between the “?” values and the “> v” values. We place t to the right of the boundary instead of to the left. FroFexperience (only), we know that the algorithm won’t reference b[k+1] or b[t-1], so there is no reason to mark those positions.

0 k t b.length

Inv b:

≤ v ? > v

**Writing the loop using the four loopy** **questions**

(1) Looking at the precondition and the postcondition, the initialization to make the invariant true is k= -1; t= b.length. (2) The loop can stop when the “?” segment is empty. That is when k = t-1. So, the loop must continue as long as k != t-1.

(3) The repetend has to make progress toward termination and keep the invariant true. Look at the middle value b[e], where e = (k+t)/2. As shown to the right. If this va

Here’s another take on binary search. Consider two “thought” array elements: b[-1] contains -∞ and b[b.length] contains ∞. Thus, we think of the array containing 3, 3, 4, and 8 as shown to the right. Of course, we must make sure that the algorithm doesn’t reference b[-1] and b[b.length], but their presence in our thoughts will make writing the algorithm easier.

-1 0 2 3 4 5 b.length

b

-∞ 3 3 4 8 ∞

Given v, we want to store a value in k that truthifies the following postcondition:

**v k**1 -1  
3 2  
5 3  
8 4  
9 4

R: b[k] ≤ v < b[k+1]

The table to the right gives examples of value v and the corresponding value of k that truthifies R, assuming that b is as given to the right above. If b is sorted and v < b[0], k is -1. If b is sorted and v occurs one or more times in b, k is the index of the rightmost occurrence of v. If b is sorted, b[0] < v, but v does not occur in b, k is the position after which v could be inserted. For example, if v = 5, k = 3: v belongs between b[3] and b[4].

We want a loop (with initialization) that truthifies R. We can truthify b[k] ≤ v by storing -1 in k. We can truthify the second relation v < b[k+1] by storing b.length-1 in k (since b[b.length] is ∞!). But we can’t do both. To break this impasse, we replace the term k+1 in R to get this invariant —we also show the relation between k and t.

Inv: b[k] ≤ v < b[t] and -1 ≤ k < t ≤ b.length

Now to the four loopy questions. First, we initially truthify the invariant by setting k to -1 and t to b.length. Second, looking at R and the invariant, we see that R will be true if the invariant is true and if t = k+1. Therefore, the loop condition is t ≠ k+1.

The repetend has to make progress toward termination and keep the invariant true. To determine how to do this, to the right, we first show the invariant and the middle index e between k and t: e = (k+t)/2. Note that because k+1 < t, we have k < e < t. This is important, because it shows that setting k or t to e will make progress toward termination.

0 k e t b.length

Inv b: -∞ ∞

≤v ? >v

Now, if b[e] ≤ v, setting k to e makes progress toward termination and keeps the invariant true; if b[e] > v, then setting t to e makes progress toward termination and keeps the invariant true. Thus, we write the algorithm as:

k= -1; t= b.length;  
 while (k+1 != t) {  
 int e= (k+t)/2;  
 if (b[e] <= v) k= e;  
 else t= e;  
 }

Note that this is the same algorithm that we wrote on the previous page, so why did we do it? To understand this, nowhere in the development do we use the fact that array b is sorted. It doesn’t have to be!

As explained above, if b *is* sorted, the algorithm finds the rightmost v in b —or the position after which v belongs if v is not in b. But if v is not sorted, it simply finds one index k that satisfies

b[k] ≤ v < b[k+1]

Thus, it is not guaranteed to find v, even if v is in b.

But this algorithm is useful in at least one place where the array is not sorted. Based on other people’s work we wrote a constant-space quicksort algorithm. It is not recursive, so there is not a bunch of stack frame