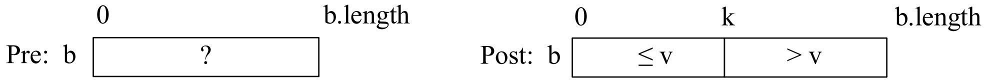
# Binary search in an array

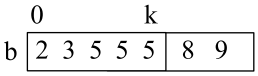
We develop an algorithm to look for a value in sorted array b. It's called *binary* search because at each iteration of its loop, it cuts the segment of b still to be searched in half, as in a dictionary search. When you search a dictionary, you don't start at the beginning and work forward. You look sort of 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. And so on.

### The specification

Given is sorted array b. We won't state in assertions that b is sorted, since b 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.



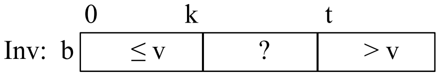
Examples: Consider array b shown to the right. If v is 5 then k = 4: the index of the rightmost 5 in b. If v is 7, k is also the index of the rightmost 5, since b[0..5] ≤ 7 and b[5+1..] > 7. In this case, one could say that v belongs after b[k].



Some people prefer the following equivalent math expression for the postcondition: b[0..k] ≤ v < b[k+1..].

### The loop invariant

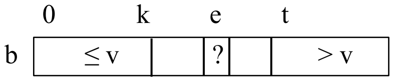
We combine the pre- and post-condition to form the invariant shown to the right. We introduced variable t to mark the boundary between the "?" values and the "> v" values. We placed t to the right of the boundary instead of to the left. From experience (only), we know that the algorithm won't reference b[k+1] or b[t-1], so there is no reason to mark those positions. Proving progress toward termination is also easier with this placement.



### Writing the loop using the four loopy questions

(1) Looking at the precondition, truthify the invariant by setting k to -1 and t to b.length. (2) The loop can stop when the "?" segment is empty, i.e. when k = t-1. Therefore, the loop continues as long as k != t-1.

(3, 4) The repetend has to make progress toward termination and keep the invariant true. Look at the middle value b[e] of the segment b[k+1..t-1], where e = (k+t)/2 (to the right). Since b is sorted, if b[e] ≤ v, then every element of b[k+1..e] ≤ v, so setting k to e keeps the invariant true and makes progress toward termination. Similarly, if b[e] > v, setting t to e keeps the invariant true and makes progress.



Thus, we write the algorithm as shown to the right. We have given the full invariant, including the bounds on k and t.

Since each iteration cuts the size of the ? segment in half, the execution time is O(log b.length).

```
k= -1; t= b.length;
// inv: -1 ≤ k < t ≤ b.length  and
//       b[0..k] ≤ v < b[t..].
while (k != t-1) {
   int e= (k+t)/2;
   // k < e < t --needed for progress
   if (b[e] <= v) k= e;
   else t= e;
}
```

### Discussion

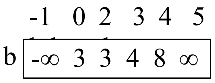
You may see binary search algorithms that terminate as soon as v is found. This algorithm is better because it gives more information: It finds the rightmost occurrence of v (if v is in the array), and if v is not in the array, it indicates where v belongs. Finally, it is not slower. Statistics show that on the average one more iteration is performed because it doesn't stop when v is first found. So one might think that it is slower. But if the loop is to stop when v is found, *two* tests are needed, on b[e] < v and b[e] = v, so each iteration may be slower.

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]. Thus, the algorithm makes sense even if the array has 0 values.

### Removing the need for b to be sorted

We redo the development of binary search with a different specification that does not require the array to be sorted, although it may not actually find v if the array is not sorted. Weird but useful.

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, our algorithm cannot reference b[-1] and b[b.length], but the presence of these thought variables allows us to change the invariant and how we think of the algorithm.



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

R: b[k] ≤ v < b[k+1]

Note that if b is not sorted, many different values of k could satisfy R, and R does not indicate at all that v is found.

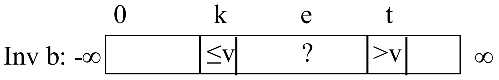
| v | k |
|---|---|
| 1 | -1 |
| 3 | 2 |
| 5 | 3 |
| 8 | 4 |
| 9 | 4 |

Suppose b is the array give above to the right, which *is* sorted The table to the right gives examples of v and the corresponding value of k that truthifies R. We discuss briefly. If v < b[0], k is -1. If v occurs one or more times in b, k is the index of the rightmost occurrence of v. If b[0] < v, but v doesn't 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 term b[k] ≤ v by storing -1 in k. We can truthify the second term 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 the following invariant —we also put bounds on 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 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 makes progress toward termination.

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 shown to the right.

```
k= -1; t= b.length;
// inv: -1 ≤ k < t ≤ b.length  and
//       b[k] ≤ v < b[t]
while (k != t-1) {
   int e= (k+t)/2;
   // k < e < t --needed for progress
   if (b[e] <= v) k= e;
   else t= e;
}
```

### Discussion

This is the same algorithm that we wrote on the previous page, so why did we do it? Nowhere in the development do we use the fact that array b is sorted. It doesn't have to be!

Of course, 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]

This algorithm is useful in at least one place where the array is not sorted. Based on other people's work, we wrote a quicksort algorithm that uses constant space! It is not recursive but iterative only, so no recursive stack frames are needed. Moreover, this binary search algorithm on an unsorted array is crucial. Fittingly, the paper was published in a book edited by A.W.Roscoe honoring Tony Hoare, the author of quicksort: "*A classical mind: essays in honour of C. A. R. Hoare*", Prentice Hall International (UK) 1994 ISBN:0-13-294844-3. You can find a preprint of this paper in the JavaHyperText entry Quicksort.