**Writing Correct Code: Binary Search**

**Loop Invariants, Bound functions, Pre and Post-conditions**

Refs: <https://reprog.wordpress.com/2010/04/30/writing-correct-code-part-3-preconditions-and-postconditions-binary-search-part-4c/>

**Invariant:**

1. An invariant is a property that remains true throughout the execution of a piece of code.
2. It’s a statement about the state of a program — primarily the values of variables - that is not allowed to become false.
3. If it becomes false, then the code is wrong.
4. Choosing the correct invariant — one that properly expresses the intent of an algorithm — is a key part of the design of code; and ensuring that the invariant remains true is a key part of the actual coding.
5. **In practice, we formalize the invariant in terms of specific vairables and values that appeared in the algorithm.**

**Bound Function:**

1. The bound function of a loop is defined as an upper bound on the number of iterations still to perform.
2. More generally, we think of it as an expression whose value decreases monotonically as the loop progresses. When it reaches zero (or drops below), we exit the loop.

**Pre-Condition and Post-Condition:**

1. A way to specify the requrements for a function (about the input and output).
2. Precondtion: Some requirement about the input that have to be true.
3. More formally: A precondition is a statement about what must be true about the inputs to a function to guarantee that the function will behave as expected(which is the post condition).

**Binary Search Problem Statement:**

Given an integer X and integers A0, A1, A2, ...,AN-1, which are presorted in ascending order, find i such that Ai = X, or return -1 if X is not in the input. There might be multiple i with Ai = X.

**Invariant For Binary Search:**

1. Informally: “If the saught value ‘*X*’ is present in the array ‘*A*’ at all, then it is present in the current range.”
2. To make it useful for deriving correct code: formalise the invariant in terms of specific variables and values.
3. And before that decide on the representation of the range under consideration.
4. There are several candidate representions: None of them greatly better or worse than the others:
5. keep track of the highest or lowest array indexes that might hold ‘*X*’, or
6. the lowest index and the size of the range ; or
7. use asymmetric indexes, where we maintain to the base of the current range and the index points past the end.
8. For this particular implementation of the rouine provided below: we choose to represent the range by two vairabels, *lower* and *upper,* which contain *X.*
9. With that representation, we can formalise the invariant as:

if *X* is at any position *i* in *A* (i.e. *a[i] == X*),

then *0 <= lower <= i <= upper.*

1. So long we ensurethis invariant is kept true, we can be confident that our coe will not fail to find *X* if it’s present.(This doesn’t show that the program will terminate)
2. Armed with this invariant, we can write the code with some
3. Code:

*int binary\_search(const A[], const int size, const int X)*

*{*

*int lower = 0;*

*int upper = size - 1;*

*/\* invariant: if a[i] == X for any i, then 0 <= lower <= i <= upper \*/*

*while(lower <= upper)*

*{*

*int i = lower + (upper - lower)/2;*

*if (X == A[i]) return i;*

*else if (X < A[i]) upper = i - 1;*

*else if (X > A[i]) lower = i ;*

*}*

*return -1;*

*}*

**Informal Proof of the Code:**

=> show the invariant hold in this three cases: Initialization, maintenance(preservation), terminaiton

1. First thing to establish: “the invariant—that will hold true for the rest of the function.”
2. So, we set the variables *lower* and *upper* to appropriate values
3. Initialization: We have ensured that the invariant is true when we first enter the loop.

Maintenance

To show that stays true throughout the running of the function, show that whenever it’s true at the top of the loop, it’s also true at the bottom of the loop (If it is true before an iteration of the loop, it remains true before the next iteration.).

1. (i) the first statement of the loop (assigning to *i*) does not affect any of the variables referenced by the invariant, so it can’t possibly cause the invariant to stop being true.

(ii) Note that *A* and *X* can never be changed as we declared them *const.*

1. What follows is a three way *IF statement*: show that the each of the three branches maintains the invariant.
2. The first branch: we have found *X,* we return out of the function.
3. *X < A[i]*: the first time we need non-trivial reasoning. As we’re in this branch: *X < A[i]*. But because *A* is sorted, *for all* *j > i, a[j] >= a[i].* This means that *X < all a[j] with j >= i.* so the highest position it can be is *A[i - 1]*.

Set upper to *i - 1*, and the invariant still holds with the new, more restrictive value of upper.

Notice what happened here: we have shrunk the range to half its previous size or less, but simple reasoning about the code persuades us that the we still know where value must be (if it’s anywhere in *A*). We are confident that we have not inadvertently excluded it from the range.

1. *X > A[i]*: follows the same reasoning: since we know that *X > A[i]*, thus *X > all A[j] where j < i*, we can conclude that the lowest position *X* can be at is *A[i + 1]*, we adjust lower accordingly and maintain the invariant.
2. Since we’ve verified that all three branches of the *IF* maintain the