Textbook Errors in Binary Searching

Richard E. Pattis
Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

This paper discusses the specification and implementation of binary searching. It begins by presenting a "standard" set of declarations, a specification, and a binary searching procedure written in Pascal. This procedure does not meet the specification: it contains five errors that also occur in many CS-1 and CS-2 textbooks. We will locate and study these errors, and show alternative procedures that correct them in a simple and understandable way.

I. Introduction: Algorithms and Procedures

Binary searching is an interesting, useful, and beautiful algorithm. Its code is simple enough to be motivated, discussed and analyzed in a beginning programming course, but it is complex enough to not be completely intuitive. Often, binary searching is the first nontrivial procedure to be annotated with preconditions, postconditions, and invariants — to help exhibit its correctness. Also, it is often the first algorithm to be presented with a nontrivial runtime analysis — using big-O notation. Although it contains code that is only slightly more complex than linear searching, binary searching offers a tremendous speed improvement for searching large arrays. And because searching is such a frequently occurring operation in software, binary search procedures are often used as subroutines in large systems.

Although the binary search algorithm is easy to state, there are many pitfalls in translating this algorithm into a specific programming language, such as Pascal. Five errors occur repeatedly in various CS-1 and CS-2 textbooks. The following sections contain a discussion of binary searching, the five common errors found in textbooks, a discussion of why they occur and how to prevent them, and finally a presentation of two versions of a "correct" binary search procedure in Pascal.

II. Declarations

We will use the constant and type declarations shown below throughout the rest of this paper, to specify an array type and its associated constants and subranges. For simplicity — and in accordance with common Pascal usage — we will assume that all the array elements are stored contiguously, starting at the index 1. With little loss of generality, we will also assume that the elements stored in the array are all of type INTEGER. Restricting array elements to be primitive types (also including PACKED arrays of characters) allows the bodies of the binary search procedures to use Pascal's built-in relational operators (e.g., <, =, and >); in the final variant of the binary search procedure (see section VI.), we will reexamine and remove this restriction.

```
1.  CONST
2.    MaxArrayIndex = 100;
3.
4.  TYPE
5.    anArrayIndex  = 1..MaxArrayIndex;
6.    anArray       = ARRAY [anArrayIndex] OF INTEGER;
7.    anArraySize   = 0..MaxArrayIndex;
```

The type anArray can store a maximum of 100 elements; doing so would fill the array. But this array may also be unfilled: it has a related variable of type anArraySize containing a value indicating the number of elements currently stored in the array; a value of 0 indicates that the array is empty.

III. Specifying Binary Search

To substantiate that certain binary search procedures do in fact contain errors, we first need an explicit statement of what each binary search procedure should accomplish: a specification of binary searching. We often supply such a statement by using a comment prefacing the procedure header; it contains the preconditions and postconditions of the procedure, stated in terms of the procedure's formal parameters. Binary searching is such a common and useful operation (and it is shown so often as an example in programming books) that some standard specification should be well established — but none is. Worse yet, many books present binary search procedures without any explicit specification of exactly what the procedure should accomplish.

Listed below is a specification for binary searching, consisting of a comment and its associated procedure header. I do not assert that this is the one true specification, but it seems to have a certain commonality with all the specifications that books present — or in the absence of such a specification, what the code they present tries to accomplish. One common variant specifies binary searching as a function that returns the index of the position of the element that is being searched for — and returns a special default value (often 0) when that element is not present in the array. Another variant returns information partially through a VAR parameter and partially through the value returned by a function. These differences are irrelevant to the content of this paper.

The comment below differentiates between two forms of preconditions: PreC and PreU. The PreC preconditions are ensured valid by the compiler or runtime system, or are checked by the procedure itself; the PreU preconditions are not checked explicitly (but must still be true for each set of actual parameters, to ensure that each procedure call operates correctly). For example, the PreC precondition 0 <= Size <= MaxArrayIndex is ensured by runtime subrange checking: because the formal parameter Size is a subrange type, anArraySize. Pascal detects a runtime error if the actual parameter is outside of the range 0..MaxArrayIndex. Alternatively, if the less-restrictive type INTEGER were used for the Size parameter, it would be inexpensive to perform an explicit check on the value of Size with an IF statement at the beginning of the procedure's body.

The PreU precondition that array A is sorted is not explicitly checked: to do so would require examining each element in the array, which would require $O(Size)$ time and nullify the speed advantage of binary searching over linear searching. Some specifications for binary searching strengthen this precondition to state that the array is in strictly ascending order, excluding the possibility of duplicate values and ensuring that at most one element in the array can be equal to Key. We omit this restriction; without it, for instance, binary searching cannot be guaranteed to return the exact same answer as linear searching.

```
1.  (***** BinarySearch
2.     PreC: 0 <= Size <= MaxArrayIndex.
3.     PreU: A is sorted in nondescending order (1 <= I <= J <= Size : A[I] <= A[J]).
4.     Post: If there exists an I such that (1 <= I <= Size) and A[I] = Key then
5.             Found = TRUE and Index = I. If no such I exists, Found = FALSE and
6.             the value of Index is undefined.
7.     Note: If Key appears in array A multiple times, then Found = TRUE and
8.             Index will be set to some arbitrary I, such that A[I] = Key.
9.     Time: O(Log Size)
10. *****)
11.
12. PROCEDURE BinarySearch (VAR A     : anArray;                (* EFFICIENCY *)
13.                         Size      : anArraySize;
14.                         Key       : INTEGER;
15.                         VAR Found : BOOLEAN;
16.                         VAR Index : anArrayIndex);
```

If any checked precondition fails, the result should be an execution error, or printing some explicit error message, or setting some error indicator. There are different opinions as to whether (and how) such failures should be reported back to the call site in a graceful manner, where further corrective action might be taken, or whether an execution error should result — terminating all the code. This interesting topic is beyond the scope of this paper. Here, we will not be concerned with the actual method of reporting the discovery of a failed precondition. But if any precondition fails to hold, BinarySearch is not required to meet its postconditions (i.e., it is not guaranteed to return a correct answer).

IV. An Incorrect Binary Search Procedure

The following Pascal procedure is a contrived combination of all the worst features of the code appearing in introductory programming texts (ACM's CS-1 curriculum) and beginning algorithm and data structures texts (ACM's CS-2 curriculum). Although this procedure may look effective at first glance, it contains five errors that we will examine in detail. These errors are logical: they are not related to machine-specific problems such as arithmetic rounding or overflow.

```
1.  PROCEDURE BinarySearch (A         : anArray;
2.                          Size      : anArraySize;
3.                          Key       : INTEGER;
4.                          VAR Found : BOOLEAN;
5.                          VAR Index : anArrayIndex);
6.  VAR Low, High : anArrayIndex;
7.  BEGIN
8.    Low := 1;
9.    High:= Size;
10.
11.   REPEAT
12.     Index:= (Low + High) DIV 2;
13.     IF Key < A[Index]
14.       THEN High:= Index-1
15.       ELSE Low := Index+1
16.   UNTIL (Low > High) OR (Key = A[Index]);
17.
18.   Found:= (Low <= High)
19. END;
```

V. The Errors

The errors discussed in this section allow `BinarySearch`, even when all its preconditions are satisfied, to (a) fail to satisfy its Time specification (Error 1); (b) produce spurious execution errors (Errors 2,4,5); or (c) produce an incorrect answer (Error 3).

Error 1: This procedure does not run in time $O(\log Size)$. The *raison d'être* for studying and using binary searching is its speed advantage (operating on large arrays) over the simpler method of linear searching. Binary searching should run in time $O(\log Size)$: if the input size is doubled, the worst-case number of operations (comparisons, arithmetic operators, array access, etc.) rises by just a constant. Yet in this `BinarySearch` procedure, the array `A` is passed as a value (not `VAR`) parameter. Virtually all Pascal compilers will transmit this parameter by first copying every element from the actual array parameter into the formal array parameter; only then will the binary searching code execute. In such a scenario, the running time is more accurately stated as $O(Size) + O(\log Size) = O(Size)$.

In an even worse scenario, some textbooks present a recursive version of `BinarySearch`, which repeatedly passes the array to be searched as a non-`VAR` formal parameter to each recursive invocation of this procedure. In this case, the entire array is copied not just once, but once for each recursive call. The running time of such code rises to $O(Size \log Size)$.

Finally, with a non-`VAR` formal parameter, the amount of extra space needed by this `BinarySearch` procedure increases from $O(1)$ to $O(Size)$; in the case of a recursive procedure, the amount of extra space may increase to $O(Size \log Size)$. On small machines — or compilers using the small machine model for code generation — passing large arrays in such a manner may cause an unexpected execution error, because of lack of available stack space.

The array `A` should be passed as a `VAR` parameter, with a special side-bar comment indicating that the reason is based purely on efficiency, not because the actual parameter is being modified. In this case, the two parameter modes available in Pascal do not allow us to specify our intent perspicuously; the semantics of the Ada `IN` mode more closely reflects our requirements.

Error 2: This procedure causes an execution error whenever `Size = 0`. In the procedure above, with `Size = 0`, the first evaluation of the expression `(Low + High) DIV 2` will result in the value `0`, which causes a subrange-out-of-bounds execution error, when it is assigned to the variable `Index`. Even if the type of `Index` were expanded to `INTEGER` — avoiding this particular error — the subsequent array access that uses `Index` (in the `IF` statement in line 13) would cause an array-out-of-bounds execution error.

For this error there are two problems to address: First, is the precondition that `Size >= 0` appropriate (instead of the more restrictive `Size > 0`)? Second, what is the underlying cause of this error and how can it be fixed?

An easy way to dismiss this error is to change the specification of `BinarySearch` to require that `PreC: Size > 0`, mandating that the array be nonempty. This new precondition can be easily implemented by changing the type of `Size` from `anArraySize` (which includes the value `0`) to `anArrayIndex` (whose lowest value is `1`). Or, we could change the definition of `anArraySize` to omit the possibility of an empty array. But a common use of binary searching is to continually build, query, and remove information from a table; such tables start empty and may become empty at various times during program execution. We shall see below, allowing for `Size = 0` does not decrease the understandability of the code; in fact, we will study a procedure that allows this weaker precondition and is also more understandable than the `BinarySearch` procedure shown above.

The underlying cause of this error is the use of the `REPEAT/UNTIL` control structure, which is inappropriate and confusing. Fundamentally, this control structure must execute its body at least once; but with an empty array, the body should be executed zero times. We could include a special `IF` test to avoid causing an error in this special case, but such extra code would only obfuscate our procedure even more, and take us further from our goal of writing an easy to understand and correct binary search procedure: these two goals are strongly linked. This simple argument alone advocates for the use of a `WHILE` control structure, which can succinctly and correctly implement binary searching. But there is even a more compelling reason underlying the inappropriateness of the `REPEAT/UNTIL` loop containing an `IF` statement.

In its simplest form, the binary search algorithm (as opposed to Pascal code) performs a trichotomous comparison between `Key` and `A[Index]`. If the two are equal, the answer is known; if they are not equal, either `Low` is increased or `High` is decreased. One reason that the `BinarySearch` procedure shown above is so difficult to understand and prove correct is that this trichotomous comparison is lexically split: the `<` and `>` tests are performed by the `IF` statement itself, but the `=` test is performed in the `UNTIL` clause. Thus, this single algorithmic concept is distributed into two control structures in the code, resulting in a lack of cohesiveness. Also, notice that because the `<` and `=` test are combined in the `ELSE` clause, either `Low` or `High` is always changed at the end of every iteration — even if `Key` has been found at the `Index` position; this nonintuitive operation contributes to Errors 3 and 5, which we study below.

Error 3: This procedure returns a wrong answer because it sets `Found` to `FALSE`, whenever the beginning of the final iteration of the `REPEAT/UNTIL` loop has `Low = High` — regardless of whether `Key` is stored in the array. Such a case, for instance, occurs whenever `Size = 1`. Because repeatedly executing the loop on larger arrays reduces the distance between `Low` and `High`, in many cases this distance eventually becomes zero, meaning `Low = High`; in all these cases `BinarySearch` may also produce an incorrect answer.

The problem is in the post-loop assignment to the parameter `Found`. Continuing our analysis from Error 2, based on the inadequacy of the `REPEAT/UNTIL` loop, we see that the value of the parameter `Index` is directly related to the local variables `Low` and `High`: whenever `Index` is used, it should always specify the midpoint between the current values of `Low` and `High`. But in this procedure, the `UNTIL` clause combines testing the new values of `Low` and `High` (one has just changed) with the value of `A[Index]`, where `Index` is computed as the midpoint of the old values of `Low` and `High`. So, these three variables are not correctly synchronized.

Thus, it is possible to terminate a loop with both `A[Index] = Key` and `Low > High`. Look at a procedure call with `Size = 1` and `Key = A[1]`. At the end of the first iteration we have `Index = 1`, and `Low = 2` because the comparison `Key < A[1]` was false, and `High = 1`. Thus, the loop exits because both `Key = A[Index]` and `Low > High`. In this case the assignment statement incorrectly sets `Found` to `FALSE`.

We can fix this problem by replacing the post-loop assignment statement by `Found:= (Key = A[Index])`. Although it is subtle to analyze, for the reasons explained above, such a procedure will always produce the correct answer. In the next section we will study binary searching procedure that uses a `WHILE` loop, which alleviates all these problems while simplifying the code too.

Error 4: This procedure causes an execution error whenever `Size` > 0 and the value of `Key` is strictly less than the element stored in `A[1]`. Notice that in this case, eventually `Low` will contain the value 1 and `High` will contain the value 1 or 2. This state will occur because the `IF` test will continually be true (because `Key` is strictly less than every element in the array), so `Low` will remain unchanged while `High` will continually be decreased until it is equal to 1 or 2. During the next iteration of the `REPEAT` loop, `Index` will receive the value 1, and once again the `IF` test in line 13 will be true. But the next assignment statement sets `High` to be 0 (which is the value of `Index-1`), and that will cause a subrange-out-of-bounds execution error, because the type of `High` is the subrange type `anArrayIndex`. An easy way to fix this error is to expand the subrange of `High` so that it also includes the value 0. See the discussion of the next error for a continued analysis in this line of reasoning.

Error 5: This procedure causes an execution error whenever `Size` = `MaxArrayIndex` and the value of `Key` is greater than or equal to the element store in `A[MaxArrayIndex]`. For reasons analogous to those in Error 4, eventually both `Low` and `High` will contain the value of `MaxArrayIndex`. During the next iteration of the `REPEAT/UNTIL` loop `Index` will receive the value `MaxArrayIndex` too, and once again the `IF` test will be false; remember that we saw at the end of the discussion of Error 2, even when `Key = A[Index]` the `ELSE` clause will be executed, changing the value of `Low`. But the next assignment statement sets `Low` to be `MaxArrayIndex+1` (which is the value of `Index+1`), and that will also cause a subrange-out-of-bounds execution error, because the type of `Low` is also the subrange type `anArrayIndex`.

This error is of a similar nature to Error 4, but it a bit more subtle and much more likely to by bypassed during testing: the conditions under which it occurs are quite restrictive. But it is not unfeasible to think of a program that must read, store in an ordered array, and continue processing exactly `MaxArrayIndex` elements; so searching a filled array for its largest element, or an element that is larger, could occur in actual programs. I would be less apt to criticize such an error if it were present in some obscure and little studied procedure; but I feel that I can legitimately hold up binary search procedures to this level of scrutiny.

In their rush to use the strictest possible subranges, some authors have forgotten that it is possible for `Low` or `High` to exceed the bounds of their "obvious" subranges. Again, an easy way to fix this error is to expand the subrange of `Low` so that it includes the value `MaxArrayIndex+1`. This type would be clumsy to specify in Pascal, because Pascal disallows constants computed from other constants; but such constants are easy to specify in Modula-2 or Ada. Unfortunately, such an approach would create an asymmetry between the `Low` and `High` variables: they would be of different types. Rather than decreasing the understandability of such code by creating new subrange types (which admittedly increases its precision), the types of both these variables should be changed to `INTEGER`; some authors define and use the subrange `0..MaxArrayIndex+1` for both.

An incorrect attempt to fix both Error 3 and Error 4 is to change the disjunction `(Low > High) OR (Key = A[Index])` into `(Low+1 >= High) OR (Key = A[Index])`. Although this restriction appears to fix the problem, disallowing `Low` and `High` from wandering outside of the `anArrayIndex` subrange, such code produces incorrect answers because the value of `Index` used in the post-loop assignment statement remains unsynchronized with the current values of `Low` and `High` — one of which just changed without the necessary recomputation of `Index`. This problem was discussed at the end of the sections explaining Errors 2, 3, and 5. Actually, it too can be overcome, but at the cost of tremendously complicating the post-loop testing code.

Finally, all the fixes described for errors 4 and 5 would be very difficult to justify and implement if the subrange type used to declare `anArray` were an enumerated type, with absolute first and last elements. In such a case, gracefully extending the subrange would be very difficult.

VI. Alternative Binary Search Procedures

In this section we examine two Pascal procedures for binary searching that do not exhibit any of the five errors discussed above. I have tried to write this code as cleanly as possible, displaying it in a style that promotes understandability. In these procedures, the number of non-blank lines of code is smaller than in any corrected version of the previously discussed binary search procedure. And, I assert that these procedures are easier to understand than the previous one; therefore it is easier to convince ourselves that this code is correct. Smaller isn't necessarily better (sometimes too much "redundant" information is removed), but it is often the case that smaller procedures are easier to understand.

```
 1.  PROCEDURE BinarySearch (VAR A     : anArray;                (* EFFICIENCY *)
 2.                          Size      : anArraySize;
 3.                          Key       : INTEGER;
 4.                          VAR Found : BOOLEAN;
 5.                          VAR Index : anArrayIndex);
 6.  VAR Low, High : INTEGER;
 7.  BEGIN
 8.    Low  := 1;
 9.    High := Size;
10.    Found:= FALSE;
11.
12.    WHILE (NOT Found) AND (Low <= High) DO BEGIN          (* still searching? *)
13.      Index:= (Low + High) DIV 2;                         (* compute midpoint *)
14.      IF        Key = A[Index]     THEN Found:= TRUE      (* trichotomy =,<,> *)
15.        ELSE IF Key < A[Index]     THEN High := Index-1   (*   found or alter *)
16.          (* IF Key > A[Index] *)  ELSE Low  := Index+1   (*   Low/High bound *)
17.    END
18.  END;
```

Note that the formal parameter `A` is declared to be a `VAR` parameter, with an appropriate side-bar comment stating why. If `Size = 0`, the body of the loop will not be executed, and the procedure will immediately return with `Found` set to `FALSE`. The parameter Found is explicitly set to `TRUE` exactly when a comparison showed `Key` to be located at `A[Index]`; in such a case, neither `Low` nor `High` change. The local variables `Low` and `High` are of type `INTEGER`, so there is no possibility of a subrange-out-of-bounds execution error (again assuming that `MaxArrayIndex` is not close to the maximum `INTEGER` value storable on a machine).

A further generalization and simplification of `BinarySearch` allows for passing an explicit trichotomy function as a parameter to this procedure. There are two advantages to this style of coding: First, in this new version of `BinarySearch` there is no dependence on using Pascal's built-in relational operators; thus, this procedure can be used to search arrays of records — the `Trichotomy` function determines how the array of records was sorted (e.g., which field or combination of fields to compare). Using this technique, `BinarySearch` can be adapted to new array types more easily — the only remaining generalization would be to change the array type, the `Key` formal parameter, and the `A,B` formal parameters of `Trichotomy` to be the same named type, such as `anElementType`. Second, this code is cleaner (it more closely resembles the algorithm): the "trichotomous" nature of this problem is now explicit in the `TrichotomyValue` type, `Trichotomy` formal parameter, and `CASE` statement.

A complete version of this procedure is shown below, including the declaration of the new enumerated type `TrichotomyValue`. Procedures as parameters are available is many Pascal implementations and all Modula-2 implementations (although in a slightly different syntactic form from the one shown below).

```
 1. TYPE TrichotomyValue = (FirstLess, BothEqual, FirstGreater);
 2.
 3.
 4. PROCEDURE BinarySearch (VAR A       : anArray;              (* EFFICIENCY *)
 5.                         Size        : anArraySize;
 6.                         Key         : INTEGER;
 7.                         VAR Found   : BOOLEAN;
 8.                         VAR Index   : anArrayIndex;
 9.                         Trichotomy : FUNCTION (A,B : INTEGER) : TrichotomyValue);
10. VAR Low, High : INTEGER;
11. BEGIN
12.   Low  := 1;
13.   High := Size;
14.   Found:= FALSE;
15.
16.   WHILE (NOT Found) AND (Low <= High) DO BEGIN          (* still searching? *)
17.     Index:= (Low + High) DIV 2;                         (* compute midpoint *)
18.     CASE Trichotomy (Key, A[Index]) OF
19.       BothEqual    : Found:= TRUE
20.       FirstLess    : High := Index-1                    (* lower High point *)
21.       FirstGreater : Low  := Index+1                    (* raise Low  point *)
22.     END
23.   END
24. END;
```

Finally, I do not assert that these procedures are correct; only that I am unable at present to find any errors in them. In a paper such as this one, which takes pot-shots at other persons code, I will gladly receive (and likewise publish) criticism that helps me better understand binary searching and its implementations.

VIII. Summary and Conclusion

Although binary searching is an easy to understand and much studied algorithm, many expositions of binary search subroutines contain errors. In this paper we have examined the specification of binary searching, catalogued and discussed five common errors, and proposed two alternative procedures that met this specification and avoided the common errors.

IX. Postscript

While writing this paper, I conducted an informal survey of twenty well known CS-1 and CS-2 textbooks (all containing binary search subroutines written in Pascal). Because of space restrictions, I was unable to include this survey in the conference proceedings; but I will hand out a copy of this survey at my talk, and I will be glad to send a copy to anyone who writes me. In summary, of these twenty books, only five had "correct" subroutines. Of the remaining sixteen, there were eleven instances of Error 1, six of Error 2, two each of Error 3 and Error 4, and one of Error 5; note that some subroutines contained multiple errors. I hope to expand the number of books surveyed by the time of the SIGCSE conference.

Finally, I hope to publish a companion paper in the near future that continues to discuss binary searching from a perspective of software engineering. This paper will include a discussion of how to avoid all execution errors in binary searching — regardless of machine arithmetic models, array subranges, and array sizes — and how to frequently detect violations of the `PreU:` precondition that the array is sorted — but not increase the procedure's complexity class.