The Binary Search revisited

The Binary search is a beautiful, simple, efficient, and —hence— well-known little algorithm for searching. Most computing scientists and programmers will be familiar with it. Unfortunately, there are two observations that cast some doubt on the level of familiarity. First, many programmers still have a relatively hard time writing down correct program code for it, this in spite of the fact that it concerns an algorithm of just a few lines. Second, most programmers believe that the algorithm only works for searching in sorted arrays, which reveals a misunderstanding. (A long time ago, we ourselves, too, used to sell the binary search to our students by first drawing an analogy with searching for a word in a dictionary. Afterwards, we learned to judge this as an educational blunder.) The purpose of this note is to remedy the situation.

*   *
*

Our starting point is that we are given a function or an array $f[a..b]$, $a<b$, of elements of some kind, such that the two outer elements $f.a$ and $f.b$ are in some relation Z to each other, a fact that we denote by $a\,Z\,b$. The problem is to find a pair of neighbouring elements of f that are in the relation Z . (Of course, such a pair need not exist; we will address this problem later on.) More precisely, we wish to construct a program with postcondition

R: $\quad a\le x \;\wedge\; x<b \;\wedge\; x\,Z\,(x+1)$

on the premise that the precondition implies

$$a<b \;\wedge\; a\,Z\,b \quad .$$

The procedure for constructing such a program is quite standard. In view of the shapes of the pre- and the postcondition we try to establish R by means of a repetition with invariants $P0$ and $P1$, given by

$P0$: $\quad a\le x \;\wedge\; x<y \;\wedge\; y\le b$

$P1$: $\quad x\,Z\,y$,

and with bound function $y-x$. Our program thus gets the form

$\{\ a<b\ \wedge\ a\,Z\,b\ \}$

$x\,,y\ :=\ a\,,b$

$\{\ \text{inv}\ \ P0\wedge P1\ \ \}\ \{\ \ \text{bnd}\ \ y{-}x\ \ \}$

$;\ \mathsf{do}\ y\neq x{+}1 \rightarrow\ \ \text{shrink}\ \ y{-}x\ \ \text{under invariance of}\ \ P0\wedge P1\ \ \mathsf{od}$

$\{\ R\ \}$.

And indeed, by construction, $P0\wedge P1\wedge y=x{+}1\ \Rightarrow\ R$.

For the shrinking of $y{-}x$ in the body of the repetition we investigate an increase of x and a decrease of y. (We do so since our problem is highly symmetric in x and y .) On the assumption that $x<h$, for some h still to be determined, assignment $x:=h$ increases x . We now find out under what circumstances this assignment maintains $P0\wedge P1$. As for the invariance of $P0$, we observe

$P0\ (x:=h)$

$\equiv$ $\{$ definition of $P0$ $\}$

$a\leq h\ \wedge\ h<y\ \wedge\ y\leq b$

$\equiv$ $\{$ $P0$ and $x<h$ are preconditions to $x:=h$,

hence $a<h\ \wedge\ y\leq b$ $\}$

$h<y$.

So we require h to satisfy not just $x<h$ but also $h<y$, and hence

$x<h\ \wedge\ h<y$.

Note Fortunately, such an h exists because from $P0$ and the guard $y\neq x{+}1$ of the repetition we conclude that $x{+}2\leq y$.
End

Having settled the invariance of $P0$ under $x:=h$, we now turn our attention to $P1$. We observe

$P1\ (x:=h)$

$\equiv$ $\{$ definition of $P1$ $\}$

$h\,Z\,y$.

Since this latter condition does not in general follow from $P1$ or $P0$, we plug it in as a guard to $x:=h$.

We next address a decrease of y . Because we already have an h such that $h<y$, we propose $y:=h$. By the problem's symmetry in x and y , our program now becomes

$\{\ a<b\ \wedge\ a\,Z\,b\ \}$

$x\,,y \;:=\; a\,,b$

$\{$ inv $\; P0 \wedge P1 \;\} \; \{ \;$ bnd $\; y{-}x \; \}$

; do $\; y \neq x{+}1 \;\; \rightarrow$

$\{ \; x{+}2 \leq y \; \}$

“ h such that $\; x < h \wedge h < y$ ”

; if $\; h\,Z\,y \rightarrow x := h$

[] $\; x\,Z\,h \rightarrow y := h$

fi

od

$\{ \; R \; \}$

Prog0

Remains the question of termination of this program. It does terminate if we can ensure that at least one of the guards of the if-statement evaluates to *true*. This will in general not be the case, but it is if relation Z satisfies

(0) $\qquad x\,Z\,y \;\Rightarrow\; h\,Z\,y \;\vee\; x\,Z\,h \qquad (\forall x, y, h)$

(The antecedent is the valid precondition $P1$; the consequent is the disjunction of the guards.)

While adopting this condition on Z , we can even ensure very fast termination by choosing h to be equal to $(x{+}y)$ div 2 , which thanks to precondition $x{+}2 \leq y$ indeed establishes $x < h \;\wedge\; h < y$. And thus, binary search is born!

* *
*

Our program computes an x satisfying postcondition R . Of course, there may be many x's satisfying R . And here we arrive at what we think is a distinguishing feature of the binary search, namely that

> it is beyond our control which x
> satisfying the postcondition
> will be generated.

(This is in sharp contrast to linear searches where a fully specified value is computed.)

In order to substantiate this feature we consider the instantiation $a\,,b \;:=\; 0\,,100$ and $Z := \mathit{true}$, meeting Prog0's precondition and (0). We then obtain

$x, y := 0, 100$

; do $y \neq x+1 \;\rightarrow$

$h := (x+y)$ $\underline{\mathsf{div}}$ 2

; if $true \rightarrow x := h$

$[\,]$ $true \rightarrow y := h$

fi

od

$\{\ 0 \leq x \ \wedge\ x < 100\ \}$,

which is a highly nondeterministic program. And the reader may check, in whatever way he likes, that, indeed, it can generate *any* value x such that $0 \leq x \ \wedge\ x < 100$.
We will return to this distinguishing feature at a later stage.

*   *
*

Meanwhile, some readers may have been puzzled by the "weird" condition (0). But it is not weird at all; when taking the contrapositive of (0), we get

(0') $\quad x\,(\neg Z)\,h \ \wedge\ h\,(\neg Z)\,y \ \Rightarrow\ x\,(\neg Z)\,y$,

and this just expresses that Z's complement relation $\neg Z$ is transitive.

Here are some of such relations Z (for array f of the appropriate type) :

$x\,Z\,y \ \equiv$

- $f.x \neq f.y$
- $f.x < f.y$
- $f.x \leq A \ \wedge\ A \leq f.y$
- $f.x * f.y \leq 0$
- $f.x \vee f.y$
- $\neg Q.x \wedge Q.y$, for any Q .

The reader can easily verify that they all satisfy (0) (or (0')).

Hint The first example —$f.x \neq f.y$— is a very appropriate choice when novices are to be introduced to the binary search.
End Hint .

$*$ $*$
$*$

The last example in the above list —$\neg Q.x \wedge Q.y$— is a frequently occurring one, and we shall now deal in more detail with its probably best-known instance, viz. $Q.i \equiv C < f.i$. That is, given

- integer array $f[a..b]$ with $a < b$
- integer constant C satisfying $f.a \le C \wedge C < f.b$,

our binary search becomes —see remark below—

$$
\begin{array}{l}
\quad \{\ a < b\ \}\ \{\ f.a \le C\ \wedge\ C < f.b\ \} \\
\quad x\,,y\ :=\ a\,,b \\
\quad \{\ \text{inv } P0\text{:}\quad a \le x\ \wedge\ x < y\ \wedge\ y \le b \\
\qquad\quad P1\text{:}\quad f.x \le C\ \wedge\ C < f.y\ \ \}\ \{\ \text{bnd } y-x\ \} \\
;\ \mathsf{do}\ \ y \ne x+1 \ \ \rightarrow \\
\qquad\qquad h\ :=\ (x+y)\ \underline{\mathsf{div}}\ 2\ \{\ a \le x < h < y \le b\ \} \\
\qquad\quad ;\ \mathsf{if}\ \ f.h\ \le\ C \rightarrow x := h \\
\qquad\qquad [\!]\ \ C\ <\ f.h \rightarrow y := h \\
\qquad\qquad \mathsf{fi} \\
\quad \mathsf{od} \\
\quad \{\ a \le x\ \wedge\ x < b\ \}\ \ \{\ f.x \le C\ \wedge\ C < f.(x+1)\ \}
\end{array}
$$

Prog1

Remark Actually, we arrived at the above program not by instantiating our original program scheme Prog0 , but by just rederiving it for the current relation $f.x \le C \wedge C < f.y$. That derivation is so simple that it can be done by heart: guided by the invariants $P0$ and $P1$, the program can be written down at once.
End Remark .

Note that the precondition of Prog0 (and hence of Prog1) is the *only* property of f used in the derivation. Thus we hope to have shown that the binary search has a right of existence outside the realm of sorted arrays.

However, if f *is* sorted in ascending order, Prog1 comes in very handy to record the presence of C in $f[a..b]$, namely by postfixing Prog1 with the assignment

$$present \ := \ (f.x = C) \quad .$$

And, in fact, it is only for the conclusion that C is absent from f — $present \equiv false$— that we use f's ascendingness.

* *
*

We have derived Prog0 (and Prog1) under a precondition — $a < b \wedge a\,Z\,b$ — . In many practical situations, however, this condition need not be satisfied. There are two general ways out. The first way out is to try to solve the problem separately for the case in which the precondition does not hold.

The more elegant way out is based on the observation that Prog1 does not inspect the array elements $f.a$ or $f.b$ at all —see precondition $a < h \ \wedge \ h < b$ to the inspection of $f.h$ — . This implies that their actual values are completely irrelevant to the (outcome of the) computation : they are thought variables mainly.

We exploit this observation in the following way. Suppose all we know about $f[a..b]$ is that it is ascending and that $a \leq b{+}1$ (the array may be empty). We define thought values $f.(a{-}1)$ and $f.(b{+}1)$ in such a way that

- $f[a{-}1 \, .. \, b{+}1]$ is ascending, and
- $f.(a{-}1) \leq C \ \wedge \ C < f.(b{+}1)$, and

since in addition $a{-}1 < b{+}1$ holds, we have thus laid down precisely the required precondition of Prog1 , be it with the instantiation $a\,,b \ := \ a{-}1\,,b{+}1$. The solution to our problem to record the presence of C in array $f[a..b]$ now reads

```
  x , y := a−1 , b+1
; do y ≠ x+1  →
        h := (x+y) div 2
        { a ≤ h ∧ h ≤ b , hence  f.h  is defined }
      ; if  f.h ≤ C → x := h
        [] C < f.h → y := h
        fi
  od
  { a−1 ≤ x  ∧  x < b+1 }  { f.x ≤ C ∧ C < f.(x+1) }
; if  a−1 = x →  { C < f.a } present := false
```

$$[\!]\ \ a{-}1<x \rightarrow \{\ a \leq x\ \wedge\ x \leq b \text{ , hence } f.x \text{ is defined } \}$$
$$present\ :=\ (f.x = C)$$

fi

The technique applied —adding thought values so as to establish a required precondition— is suitable in many other examples as well.

*   *
*

We conclude this note on the binary search with two exercises and a question. The exercises are

(i) Given that integer array $f[a..b]$, $a \leq b$, is the concatenation of an increasing and a decreasing sequence, find the top (maximum). More precisely, design a program that computes a value M such that

$$\begin{array}{ll} & a \leq M \wedge M \leq b \\ \wedge & f[a\ ..\ M\ ] \text{ is increasing} \\ \wedge & f[M\ ..\ b\ ] \text{ is decreasing} \end{array}$$

(Note : $f[a\ ..\ M\ ]$ is increasing $\equiv\ \langle\ \forall i\colon\ a \leq i < M\colon\ f.i < f.(i{+}1)\ \rangle$)
.

(ii) Given that integer array $f[a\ ..\ b\ ]$, $a < b$,

- is the concatenation of two increasing sequences
- contains a dip, i.e.

$$\langle\ \exists i\colon\ a \leq i < b\colon\ f.i > f.(i{+}1)\ \rangle \quad , \quad \text{and}$$

- satisfies $f.a > f.b$,

design a program to compute a dip .

Both exercises can be solved using a binary search.

If, however, in exercise (ii) the given $f.a > f.b$ is dropped, we no longer know of a way to solve this problem with the binary search technique, not even if we introduce a thought value $f.(a{-}1)$ such that $f.(a{-}1) > f.b$. The reason why the introduction of such a thought value does not help here is that it may introduce a second dip at $a{-}1$, and as we said before, it is now beyond our control which dip will be computed by the binary search : it could be the required one, but it could also be the thought dip that we ourselves introduced.

The as yet unsolved problem that remains is to identify the *precise* conditions for a binary search to be applicable.