# Notes on Sequential Program Verification

**Jayadev Misra**                                             **2/26/98**

We discuss the method of *Inductive Assertions* introduced by Floyd and refined by Hoare, and others. We will study the following idealized problem: Given a program written in a simple imperative language without input/output commands or procedure calls, is it correct? It is possible to introduce input/output, procedure calls and many other features for the programming language, but we will restrict ourselves to the following constructs:

- Assignment statement: This is of the form, $x := e$, where $x$ is a simple variable and $e$ is an expression. We will introduce arrays later.

- Sequencing: This is of the form, $s1; s2$, where $s1$ and $s2$ are statements.

- Conditional: This is of the form, **if** $b$ **then** $s1$ **else** $s2$ **fi** , where $s1$ and $s2$ are statements and $b$ is a predicate.

- Loops: This is of the form, **while** $b$ **do** $s$ **od** , where $s$ is a statement and $b$ is a predicate.

**Program Specification:** You are given an input specification (constraints/assertions on the values of variables before execution of the program) and an output specification (constraints/assertions on the values of variables after completion of the program).

**Examples**

- input specification: $x = X$, output specification: $x = sqrt(X)$

- input specification: file $f$ has zero or more records,
  output specification: records in $f$ are sorted in increasing order.

- input specification: file $f$ has zero or more records and $f = F$,
  output specification: records in $f$ are sorted in increasing order and $f$ is a permutation of $F$.

We need a formal language to write the input, output specifications. We will consider that later.

The verification problem is to establish: given that the input assertions are met before execution of the program then the program terminates, and the output assertions are met at termination. Thus, we have two separate proofs to carry out: (1) the proof of correctness *assuming* that the program terminates, (2) the proof of termination. Traditionally, the proof of (1) is called a proof of *partial correctness* and proof of both (1,2) is called a proof of *total correctness*.

There are two different views of verification. Analytic or a-posteriori method calls for a correctness proof given a program and its specification. Synthetic or

a-priori view is to design a program along with its correctness proof from its specification. I will illustrate the analytic method in this note, though most of our subsequent work will be along the other line.

# 1 Mechanics of Proof Construction

Given a program and its specification,

1. Invent assertions and attach them to specific program points. Intuitively, choose assertions such that they hold whenever control reaches the corresponding program point. Attach input/output assertions to the beginning and end of the program.

   There is no general technique and few guidelines for this step.

2. Prove that the assertions indeed hold whenever control reaches the corresponding program point.

   There is a technique that converts this problem to proving some logical propositions over the data types of the program.

3. Prove program termination.

   There is a technique that converts this problem to proving some logical propositions over the data types of the program.

We will study the techniques referred to in (2,3) above at some length; a few heuristics are given for (1).

**Example** Consider

$$
\begin{aligned}
&\textbf{while}\ \ v \neq 0\ \ \textbf{do} \\
&\quad u := u + 1; \\
&\quad v := v - 1 \\
&\textbf{od}
\end{aligned}
$$

Given input assertion, $u, v = A, B$ and output assertion $u = A + B$, we will attempt to prove this program as follows. First, we invent the assertion $u + v = A + B$ and attach it to the loop so that the assertion holds each time before the test in the loop is evaluated. Then, we will show that this proposition indeed holds whenever control reaches the point where the test in the loop is evaluated. Next, we observe that the exit from the loop requires $v = 0$, which coupled with $u + v = A + B$ establishes $u = A + B$. However, this fact holds only if the loop terminates. As you have probably observed by now, the loop may not terminate in all cases, for instance, if $v < 0$ before the program is started. Therefore, we modify the input specification to $u, v = A, B \wedge v \geq 0$, and reattempt the proof.

**Annotation**  We write $\{p\}\ s\ \{q\}$ to stand for: if $p$ holds before the execution of $s$ and $s$ terminates (starting in any state that satisfies $p$) then $q$ holds at termination.

## 2  Assignment Statement

Let $q[x := e]$ be the predicate obtained from $q$ by replacing all occurrences of $x$ by $e$. You can prove

> $\{p\}\ x := e\ \{q\}$ by showing that
> $p \Rightarrow q[x := e]$

**Examples**

1.  $\{y > 10\}\ x := y\ \{x > 5\}$ is proved by showing
    $y > 10 \Rightarrow y > 5$

2.  $\{y > 10\}\ y := x\ \{x > 5\}$ is proved by showing
    $y > 10 \Rightarrow x > 5$

    Since you can't prove the above formula, you can't prove the specification as given.

3.  $\{x \geq 0\}\ x := x + 1\ \{x > 0\}$ is proved by showing
    $x \geq 0 \Rightarrow x + 1 > 0$

4.  $\{y = 10\}\ x := y\ \{y > 5\}$ is proved by showing
    $y = 10 \Rightarrow y > 5$

    Observe that the assignment to $x$ has no effect on the value of $y$; this means assignments have no side effects.

5.  (A small design problem)
    What should $e$ be such that the following holds?

    $$\{sum = (+j:\ 0 \leq j < n:\ A[j])\}$$
    $$sum := e$$
    $$\{sum = (+j:\ 0 \leq j \leq n:\ A[j])\}$$

    Applying the axiom of assignment, we have to show,

    $$[sum = (+j:\ 0 \leq j < n:\ A[j])]\ \Rightarrow [e = (+j:\ 0 \leq j \leq n:\ A[j])]$$

    i.e., we have to show

    $$[sum = (+j:\ 0 \leq j < n:\ A[j])]\ \Rightarrow [e = sum + A[n]]$$

    We can establish this if $e$ is $sum + A[n]$. So, our assignment is $sum := sum + A[n]$.

**Exercise** Given the statement $s := false$ and the post-condition $(b \equiv s \wedge t) \wedge (s \vee t)$ compute the precondition.

# 3 Consequence

This rule says that in order to prove $\{p\}\ s\ \{q\}$ it is sufficient to prove $\{p'\}\ s\ \{q'\}$ where $p \Rightarrow p'$ and $q' \Rightarrow q$. That is, once you have $p \Rightarrow p'$ and $q' \Rightarrow q$ you can strengthen the lhs of $\{p'\}\ s\ \{q'\}$ (to $p$) and/or weaken the rhs (to $q$). This rule applies for any program $s$. Thus, given

$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$
we can conclude, by lhs strengening,
$\{x > 5\}\ x := x + 1\ \{x > 0\}$
and by rhs weakening
$\{x > 5\}\ x := x + 1\ \{x \geq 0\}$

We have already used this rule in treating the assignment statement. The assignment axiom tells us
$\{q[x := e]\}\ x := e\ \{q\}$
Thus, to prove $\{p\}\ x := e\ \{q\}$, first apply the assignment axiom and then the lhs strengthening rule, i.e.,
$p \Rightarrow q[x := e]$

# 4 Statement Sequences

To prove $\{p\}\ s1; s2\ \{q\}$ you have to find a predicate $r$ such that
$\{p\}\ s1\ \{r\}$, and $\{r\}\ s2\ \{q\}$
This rule applies for any two programs, $s1$ and $s2$, not just assignments. You can repeat this rule to prove facts about any sequence of statements, not just of length 2. Thus, to show
$\{p\}\ s1;\ s2;\ s3\ \{q\}$
you have to find $r, u$ such that
$\{p\}\ s1\ \{r\}$, and $\{r\}\ s2\ \{u\}$ and $\{u\}\ s3\ \{q\}$
In most cases, you don't have to guess the intermediate assertions. In particular, in proving $\{p\}\ s1;\ s2\ \{q\}$ if $s2$ is an assignment statement then your task is easy. Let us say $s2$ is $x := e$. For $q$ to hold as a post-condition of $s2$, we know that $q[x := e]$ has to hold as a pre-condition. So, all we have to show is,
$\{p\}\ s1\ \{q[x := e]\}$
For a sequence of assignment statements, apply this idea over and over to push the post-condition all the way to the back.

**Example** The following program exchanges $x, y$.
$\{x, y = A, B\}\ t := x;\ x := y;\ y := t\ \{x, y = B, A\}$
Pushing the post-condition before $y := t$, we get
$\{x, y = A, B\}\ t := x\ ; x := y\ \{x, t = B, A\}$

4

Similarly, pushing the post-condition again
$$\{x, y = A, B\} \ t := x \ \{y, t = B, A\}$$
Finally, pushing it one last time, our proof obligation is
$$x, y = A, B \Rightarrow y, x = B, A$$

**Exercise**   Prove the following program, where $x, y$ are integers.
$$\{x, y = A, B\} \ x := x + y; \ y := x - y; \ x := x - y \ \{x, y = B, A\}$$

**Exercise**   Prove the following program, where $x, y$ are bit-strings and $\oplus$ is the element-wise exclusive-or operation on bit-strings.
$$\{x, y = A, B\} \ x := x \oplus y; \ y := x \oplus y; \ x := x \oplus y \ \{x, y = B, A\}$$

# 5   Conditional

To prove $\{p\}$ **if** $b$ **then** $s1$ **else** $s2$ **fi** $\{q\}$, show the following
$$\{p \wedge b\} \ s1 \ \{q\} \ , \ \text{and} \ \ \{p \wedge \neg b\} \ s2 \ \{q\}$$
Again, $s1$ and $s2$ can be any program.

**Example**   Show that
$$\{true\} \ \textbf{if} \ x \geq 0 \ \textbf{then} \ y := x \ \textbf{else} \ y := -x \ \textbf{fi} \ \{y \geq 0 \wedge x^2 = y^2\}$$
We have to show that
$$\{x \geq 0\} \ y := x \ \{y \geq 0 \wedge x^2 = y^2\} \ \text{and,}$$
$$\{x < 0\} \ y := -x \ \{y \geq 0 \wedge x^2 = y^2\}$$
We prove the above two using the rules for assignment:
$$x \geq 0 \Rightarrow x \geq 0 \wedge x^2 = x^2$$
$$x < 0 \Rightarrow -x \geq 0 \wedge x^2 = (-x)^2$$
These are, finally, formulae in logic (and arithmetic) that you can prove by using the properties of numbers.

Be extremely careful about the **if-then** statement. If you have to prove
$$\{p\} \ \textbf{if} \ b \ \textbf{then} \ s \ \textbf{fi} \ \{q\}$$
then you have to show
$$\{p \wedge b\} \ s \ \{q\}$$
But that is not enough! You must prove the **else** branch, too. It is empty but requires you to show $\{p \wedge \neg b\}$ do nothing $\{q\}$. That is,
$$p \wedge \neg b \ \Rightarrow \ q$$
We can do a backward substitution over **if** $b$ **then** $s1$ **else** $s2$ **fi** as follows. Given that

$$\{u\} \ s1 \ \{q\} \ \text{and}$$
$$\{v\} \ s2 \ \{q\}$$

a pre-condition of **if** $b$ **then** $s1$ **else** $s2$ **fi** given that $q$ is a post-condition is

$$(b \Rightarrow u) \ \wedge \ (\neg b \Rightarrow v)$$

**Exercise**  Write and prove correctness of a program that stores the maximum of $a, b$ and $c$ in $m$.

**Exercise**  For the above problem the following program has been suggested. Prove its correctness.

> **if** $a > b$ **then** $n := a$ **else** $n := b$ **fi** ;
> **if** $n > c$ **then** $m := n$ **else** $m := c$ **fi**

**Exercise**  Write and prove correctness of a program that determines if three given positive real numbers specify the lengths of a triangle's sides.

# 6   Loops

You will work with specifications of the form $\{p\}$ **while** $b$ **do** $s$ **od** $\{q\}$. To prove that this specification is met you have to postulate a loop invariant.

1. A loop invariant, $I$, is a predicate that satisfies
   $$\{I \wedge b\}\ s\ \{I\}$$
   You have to do three other things:

2. Show that $I$ holds before you start the loop,
   $$p \Rightarrow I.$$

3. Show that $q$ holds when you end the loop,
   $$I \wedge \neg b \Rightarrow q$$

4. Loop terminates. We take this up later.

**Example**  Consider

> $\{x, y = A, B\}$
> $z := 0;$
>   **while** $y \neq 0$ **do**
>     $z := z + x;$
>     $y := y - 1$
>   **od**

For the loop we will have to find an expression whose value is the same at the beginning of each iteration. What is the unchanging relationship among $x, y$ and $z$? Try $z + x * y = A * B$ as the invariant, $I$. Now, we have to show:

1. $\{z + x * y = A * B \ \wedge \ y \neq 0\}\ z := z + x;\ y := y - 1\ \{z + x * y = A * B\}$

2. $\{x, y = A, B\}\ z := 0\ \{z + x * y = A * B\}$

3. $z + x * y = A * B \ \wedge \ y = 0 \ \Rightarrow \ z = A * B$

4. Loop terminates.

I find it more convenient to postulate the invariant, prove (2,3) and then prove that $I$ is indeed an invariant (i.e., the formula in 1 holds). After all, the work involved in proving the formula in (1) is substantially more than in (2) or (3); the latter ones are simple logical formulae. If you fail in (2) or (3) after proving (1), you will have wasted a lot of effort.

**Exercise**   The following program is alleged to compute the greatest common divisor ($gcd$) of variables $m, n$. Prove its correctness (skip termination).

$\{m > 0, \ n > 0, \ gcd(m, n) = G\}$
  **while** $\ m \neq n\ $ **do**
    **if** $\ m > n\ $ **then** $\ m := m - n$
    **else** $\ n := n - m$
    **endif**
  **od**
$\{m = G\}$

**Hint**   Try $m > 0, \ n > 0, \ gcd(m, n) = G$ as the invariant.

**Exercise**   Design proof rules for the following programming constructs.

  **repeat**
   $S$
  **until** $\ b$
  **end**

First $S$ is executed, then $b$ is tested. If $b$ is false then the construct is repeated, else the repeat loop terminates. This differs from do-while in that $S$ is executed at least once in the repeat loop.

# 7   Program Annotation

An annotated program is a program mixed with assertions; assertions are attached to specific points in the program text. An annotation represents your guesses about a program; if you attach an assertion $p$ to a point in the program text you are guessing that $p$ holds whenever the control reaches that point. After annotating a program you have to prove your guesses. If you are careful, your guesswork and proof work can be minimized. Follow these rules:

- If you have an assignment statement, $x := e$ with a post-condition $q$, write $q[x := e]$ as a pre-condition.

- If you have a conditional, **if** $\ b\ $ **then** $\ s1\ $ **else** $\ s2\ $ **fi** , with a pre-condition $p$ and a post-condition $q$, write $p \wedge b$ as the pre-condition for $s1$ and $p \wedge \neg b$ for $s2$, and $q$ as the post-condition for both.
  Note: It is best to treat **if** $\ b\ $ **then** $\ s\ $ **fi** , as the statement **if** $\ b\ $ **then** $\ s$ **else** $\ $ **skip** $\ $ **fi** , writing an empty **else** -clause explicitly.

- If you have a loop, guess an invariant, $I$ and annotate as follows.

$$\{I\}$$
$$\quad \textbf{while} \ \ b \ \ \textbf{do}$$
$$\quad\quad \{I \wedge b\} \ s \ \{I\}$$
$$\quad \textbf{od}$$
$$\{I \wedge \neg b\}$$

After the program is completely annotated you have to show that for two assertions $\{p\}$ and $\{q\}$ that are next to each other (without intervening program fragments), $p \Rightarrow q$.

**Example**   Let me describe in full detail how I annotate a simple program. You are given the following specification — regard it as a partial annotation — for a program used to multiply $x$ and $y$ and store the result in $z$.

$$\{x * y = t\}$$
$$z := 0;$$
$$\quad \textbf{while} \ \ y \neq 0 \ \ \textbf{do}$$
$$\quad\quad \textbf{if} \ \ odd(y) \ \ \textbf{then} \ \ z := z + x \ \ \textbf{fi} \ ;$$
$$\quad\quad y := y \div 2;$$
$$\quad\quad x := x + x$$
$$\quad \textbf{od}$$
$$\{z = t\}$$

We can't take the initial assertion and move it forward over the statement $z := 0$. So, we really can't apply any of the rules except guessing an invariant for the loop. Let me guess –later, I will tell you how I guessed– the invariant: $z + x * y = t$. So, I get the annotation

$$\{x * y = t\}$$
$$z := 0;$$
$$\{z + x * y = t\}$$
$$\quad \textbf{while} \ \ y \neq 0 \ \ \textbf{do}$$
$$\quad\quad \{z + x * y = t \wedge y \neq 0\}$$
$$\quad\quad \textbf{if} \ \ odd(y) \ \ \textbf{then} \ \ z := z + x \ \ \textbf{fi} \ ;$$
$$\quad\quad y := y \div 2;$$
$$\quad\quad x := x + x$$
$$\quad\quad \{z + x * y = t\}$$
$$\quad \textbf{od}$$
$$\{z + x * y = t \wedge y = 0\}$$
$$\{z = t\}$$

Now, we can push the invariant through the statement $z := 0$. Also, we can make some headway inside the loop: We can push the invariant from the bottom of the loop through the two assignment statements; also, we can push it forward through the conditional statement.

$$\{x * y = t\}$$
$$\{0 + x * y = t\}$$
$$z := 0;$$
$$\{z + x * y = t\}$$
**while** $y \neq 0$ **do**
$\quad\{z + x * y = t \wedge y \neq 0\}$
$\quad$**if** $odd(y)$ **then** $\{z + x * y = t \wedge y \neq 0 \wedge odd(y)\}$
$\quad\quad\{z + x + (x + x) * (y \div 2) = t\}$
$\quad\quad z := z + x$
$\quad\quad\{z + (x + x) * (y \div 2) = t\}$
$\quad$**else** $\{z + x * y = t \wedge y \neq 0 \wedge \neg odd(y)\}$
$\quad\quad\{z + (x + x) * (y \div 2) = t\}$
$\quad$**fi** ;
$\quad\{z + (x + x) * (y \div 2) = t\}$
$\quad y := y \div 2;$
$\quad\{z + (x + x) * y = t\}$
$\quad x := x + x$
$\quad\{z + x * y = t\}$
**od**
$\{z + x * y = t \wedge y = 0\}$
$\{z = t\}$

The part dealing with program proving is over. Now we have to prove the following propositions that we get from the adjacent assertions in the program. We can use algebra, arithmetic and logic in proving these propositions.

$$x * y = t \Rightarrow 0 + x * y = t$$
$$z + x * y = t \wedge y \neq 0 \wedge odd(y) \Rightarrow z + x + (x + x) * (y \div 2) = t$$
$$z + x * y = t \wedge y \neq 0 \wedge \neg odd(y) \Rightarrow z + (x + x) * (y \div 2) = t$$
$$z + x * y = t \wedge y = 0 \Rightarrow z = t$$

Invariably, the biggest hurdle and the most creative part is in finding the right invariant. There are no accepted heuristics. The following two could be useful.

- Replace some constant by a variable in the post-condition to be proven.

- If the post-condition is of the form $z = f(A, B, C, D..)$, where $A, B, C, D..$ are the initial values of $x, y, u, v..$ then try as invariant, $z \oplus f(x, y, u, v, ..) = f(A, B, C, D..)$, where $\oplus$ is the operation you are applying on $z$ ($\oplus$ was $+$ in the last example).

**Example** The goal is to compute the number of ones in the binary representation of a natural number, $n$. The result is computed in $c$. Let $ones$ be a function defined as follows:
$\quad ones(0) = 0$

$ones(2x) = ones(x)$
$ones(2x + 1) = 1 + ones(x)$

The program is as follows.

$\{N = n\}$
$c := 0;$
  **while** $n \neq 0$ **do**
    **if** $odd(n)$ **then** $c := c + 1$ **fi** ;
    $n := n \div 2$
  **od**
$\{c = ones(N)\}$

As before, we postulate an invariant. Using heuristic (2) try: $c + ones(n) = ones(N)$. We get the annotation

$\{N = n\}$
$\{0 + ones(n) = ones(N)\}$
$c := 0;$
$\{c + ones(n) = ones(N)\}$
  **while** $n \neq 0$ **do**
    $\{c + ones(n) = ones(N) \wedge n \neq 0\}$
    **if** $odd(n)$ **then**
      $\{c + ones(n) = ones(N) \wedge n \neq 0 \wedge odd(n)\}$
      $\{c + 1 + ones(n \div 2) = ones(N)\}$
      $c := c + 1$
      $\{c + ones(n \div 2) = ones(N)\}$
    **else** $\{c + ones(n) = ones(N) \wedge n \neq 0 \wedge \neg odd(n)\}$
      $\{c + ones(n \div 2) = ones(N)\}$
    **fi** ;
    $\{c + ones(n \div 2) = ones(N)\}$
    $n := n \div 2$
    $\{c + ones(n) = ones(N)\}$
  **od**
$\{c + ones(n) = ones(N) \wedge n = 0\}$
$\{c = ones(N)\}$

The logical formulae that have to be proven are,

$N = n \Rightarrow 0 + ones(n) = ones(N)$
$c + ones(n) = ones(N) \wedge n \neq 0 \wedge odd(n) \Rightarrow c + 1 + ones(n \div 2) = ones(N)$
$c + ones(n) = ones(N) \wedge n \neq 0 \wedge \neg odd(n) \Rightarrow c + ones(n \div 2) = ones(N)$
$c + ones(n) = ones(N) \wedge n = 0 \Rightarrow c = ones(N)$

The first one is trivial to prove. For the other three, you need the properties of *ones* for your proof.

# 8  Termination

To show that a program terminates for a given pre-condition, it is sufficient to show that each loop in it terminates. So, we concentrate on terminations of loops. Suppose you have already shown that for the program, **while** $b$ **do** $s$ **od** with pre-condition $p$, predicate $I$ is a loop invariant. To prove termination postulate an integer-valued *variant function*, $d$, defined on program variables, and show that

- the value of $d$ is non-negative, i.e.
    $$I \Rightarrow d \geq 0$$
    (You need to have the right $I$ for this task. You may have to go back and prove that $I \wedge d \geq 0$ is a loop invariant; then $d \geq 0$ follows from $I \wedge d \geq 0$.)

- each iteration of the loop decreases the value of $d$. You may find it necessary to use the loop invariant for this proof, i.e., you may have to show
    $$\{I \wedge b \wedge d = k\} \ s \ \{d < k\}$$

**Example**   Show that the following program terminates under the pre-condition $i \leq j$. Assume $i, j$ are integers.

> **while** $i \neq j$ **do**
>   **if** $b$ **then** $i := i + 1$ **else** $j := j - 1$ **fi**
> **od**

First, observe that $i \leq j$ is a loop invariant and it holds initially. (Exercise: Prove it.) Let us postulate $j - i$ as the variant function. Then, we have to show

$i \leq j \Rightarrow j - i \geq 0$ and,

$\{j - i = k \wedge i \neq j\}$
    **if** $b$ **then** $i := i + 1$ **else** $j := j - 1$ **fi**
$\{j - i < k\}$

Exercise: Complete the proof.

Exercise: Are the following functions possible variant functions for this example: $j - i + 1$, $j^2 - i^2$, $j$?

**Example**   The following example is due to Carroll Morgan. The variant function chosen in this example is simple, but it does not give a very useful upper bound on the number of iterations.

Given is a matrix of integers. A *toggle* operation on a column replaces each element $x$ of the given column by $-x$. Similarly, *toggle* operation may be applied to a row. Henceforth, a *line* means either a row or a column. Show a strategy by which the sum of elements in every line becomes non-negative, by repeatedly applying toggles.

The problem asks for a terminating program so that at termination, the sum of the elements of each line is non-negative. Use the following strategy: as long as there is a line whose sum is negative apply toggle to that line (if there are several such lines, choose one arbitrarily). We show that this program terminates, and at termination each line sum is non-negative.

To prove termination, Let $N$ be the sum of the absolute values of all elements in the matrix and $s$ the sum of all elements at any step. We show that $N - s$ decreases in each step. A toggle operation does not affect $N$. And, a toggle operation replaces a line with a negative sum by one with a positive sum; so, $s$ strictly increases. Therefore, $N - s$ decreases. Since $N - s$ is a non-negative integer, it can only decrease a finite number of times. Therefore, the program terminates. At termination, there is no line with negative sum (otherwise, the program would not terminate).

# 9 Program Design

Here is an example of program design starting from a specification. You are given an array in which the values are ascending, i.e., we have an array $A$, consisting of elements $A[0]...A[N]$, where

(G0): $0 < N \ \wedge \ A[0] \leq ... \leq A[N]$.

We are also given a value $v$ that lies between the array minimum and maximum:

(G1): $A[0] \leq v < A[N]$.

Write a program that establishes

(G2): $0 \leq i < N \ \wedge \ A[i] \leq v < A[i+1]$.

To start the design, get a loop invariant: Use the heuristic to repace $i + 1$ by $j$.

(P): $0 \leq i < N \ \wedge \ A[i] \leq v < A[j]$.

This can be established initially by having (see G1)

$i := 0; \ j := N$

How do we meet the desired postcondition? Set the loop iteration condition to $i + 1 \neq j$. So, we have the program skeleton:

$\{0 < N \ \wedge \ A[0] \leq v < A[N]\}$
$\quad i := 0; \ j := N;$
$\{0 \leq i < N \ \wedge \ A[i] \leq v < A[j]\}$
$\quad \textbf{while} \ \ i + 1 \neq j \ \ \textbf{do}$

$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j] \;\wedge\; i+1 \ne j\}$$
$$\text{S}$$
$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j]\}$$
**od**
$$\{0 \le i < N \;\wedge\; A[i] \le v < A[i+1]\}$$

To construct S, introduce $h$, where $i \le h < j$. S becomes

$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j] \;\wedge\; i+1 \ne j \;\wedge\; i \le h < j\}$$

S:: **if** $A[h] \le v$ **then** S' **else** S" **fi**

$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j]\}$$

Now, we have two program design tasks. Design S' and S" such that:

$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j] \;\wedge\; i+1 \ne j \;\wedge\; i \le h < j \;\wedge\; A[h] \le v\}$$
$$\text{S'}$$
$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j]\}, \text{ and}$$

$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j] \;\wedge\; i+1 \ne j \;\wedge\; i \le h < j \;\wedge\; A[h] > v\}$$
$$\text{S"}$$
$$\{0 \le i < N \;\wedge\; A[i] \le v < A[j]\}$$

It is not hard to see that the following designs for S' and S" are sufficient.

S':: $i := h$, and
S":: $j := h$

# 10 Linear Search; an example with Arrays

We have an array $A$, consisting of elements $A[0]...A[N-1]$, where $N > 0$. You are asked to search for a value $v$ in $A$. Set a boolean variable $found$ to $true$ if $v$ is in $A$; set it $false$, otherwise.

The simplest strategy is linear search. Let $i$ show how far we have already searched.

```
{N > 0}
found := false;  i := 0;
 while  ¬found ∧ i < N  do
    found := (A[i] = v);
    i := i + 1
 od
 {found = (∃j : 0 ≤ j < N : A[j] = v)}
```

The post-condition says that $found$ is $true$ iff $v$ is in $A$.

Let us use heuristic (1) to get a loop invariant.

$\quad I :: found = (\exists j : 0 \le j < i : A[j] = v)$

Now, a quick check to see that this invariant holds before we start the loop and that it will give us the required post-condition upon loop termination.

For the initial condition part, pushing $I$ through the first two initialization statements gives us

$\quad false = (\exists j : 0 \le j < 0 : A[j] = v)$

The right side of the above evaluates to *false* because we have existential quantification over empty range. Great, so far!

Next, let us check the post-condition. We have to show,

$\quad I \wedge \neg(\neg found \wedge i < N) \Rightarrow (\exists j : 0 \le j < N : A[j] = v)$

The antecedent of the implication can be simplified to

$\quad (I \wedge found) \vee (I \wedge i \ge N)$

So, the proof obligation is

$\quad (I \wedge found) \Rightarrow (\exists j : 0 \le j < N : A[j] = v)$, and
$\quad (I \wedge i \ge N) \Rightarrow (\exists j : 0 \le j < N : A[j] = v)$

The first one is easy to see. But the second one is not even true! The trouble is that we can't deduce that $i = N$ from the antecedent; all that we can deduce is $i \ge N$. The way out is to embellish the invariant to make it stronger, by adding the conjunct $0 \le i < N$,

$\quad J :: (0 \le i < N) \wedge found = (\exists j : 0 \le j < i : A[j] = v)$

You should now do the annotation of the program and derive verification conditions before reading any further.

$\{N > 0\}$
$\{(0 \le 0 \le N) \wedge [false = (\exists j : 0 \le j < 0 : A[j] = v)]\}$
$\quad found := false;$
$\{(0 \le 0 \le N) \wedge [found = (\exists j : 0 \le j < 0 : A[j] = v)]\}$
$\quad i := 0;$
$\{(0 \le i \le N) \wedge [found = (\exists j : 0 \le j < 0 : A[j] = v)]\}$
$\quad\quad \textbf{while} \ \neg found \wedge i < N \ \textbf{do}$
$\{(0 \le i \le N) \wedge [found = (\exists j : 0 \le j < 0 : A[j] = v)] \wedge \neg found \wedge i < N\}$
$\{(0 \le i + 1 \le N) \wedge [(A[i] = v) = (\exists j : 0 \le j < i + 1 : A[j] = v)]\}$
$\quad\quad found := (A[i] = v);$
$\{(0 \le i + 1 \le N) \wedge [found = (\exists j : 0 \le j < i + 1 : A[j] = v)]\}$
$\quad\quad i := i + 1;$
$\{(0 \le i \le N) \wedge [found = (\exists j : 0 \le j < i : A[j] = v)]\}$
$\quad\quad \textbf{od}$
$\{(0 \le i \le N) \wedge [found = (\exists j : 0 \le j < i : A[j] = v)] \wedge \neg(\neg found \wedge i < N)\}$
$\{found = (\exists j : 0 \le j < N : A[j] = v)\}$

Next, let us prove termination of the loop. In each iteration, the unscanned portion of the loop is decreasing. This suggests that we try the variant function,

$N - i$. I will leave you to show that,

$J \Rightarrow (N - i) \geq 0$, and

the value of $N - i$ decreases in each iteration of the loop.

## A Simpler Linear Search

In the last algorithm we performed two checks in each iteration: whether the item $v$ has been found and if the array has been completely scanned. Now, if we are guaranteed to find the item – i.e., $v \in A$ is a pre-condition – then we need not check if the array has been completely scanned. Of course, there is not much point in setting $found$; we know it will be $true$.

$\{N > 0, (\exists k : 0 \leq k < N : A[k] = v)\}$
$i := 0;$
  **while** $A[i] \neq v$ **do**
    $i := i + 1;$
  **od** ;
$\{N > 0, A[i] = v)\}$

A technique to ensure that $v \in A$ is to add $v$ to $A$. So, we first store $v$ in $A[N]$ and then search for it using the method shown above, because we are sure to find it. If the search yields a value other than $N$ then we set $found$ to $true$, otherwise to $false$. We use the notation $v \in A[0. < N]$ to mean $(\exists j : 0 \leq j < N : v = A[j])$. The program is

$\{N > 0\}$
$A[N] := v; \ i := 0;$
  **while** $A[i] \neq v$ **do**
    $i := i + 1$
  **od** ;
$found := (i < N)$
$\{found = v \in A[0. < N]\}$

What invariant should we use? As before, let us use $0 \leq i \leq N$ as one of the conjuncts. Another conjunct is that we have not seen $v$ in the segment $A[0]...A[i - 1]$. That is $(v \notin A[0. < i])$. So we postulate,

$K :: 0 \leq i \leq N \wedge (v \notin A[0. < i])$

Let us quickly check that $K$ holds after the initialization and that $K$ can be used to establish the post-condition. Pushing $K$ backward through the initialization, we get $0 \leq 0 \leq N \wedge (v \notin A[0. < 0])$. The first conjunct holds, given that $N > 0$. The second conjunct is $true$ (prove it). For the post-condition, we have $K \wedge A[i] = v$. We are required to show that after completing execution of $found := (i < N)$ the predicate, $found = (v \in A[0. < N])$ holds. Pushing this predicate through the assignment to $found$, we get $(i < N) = (v \in A[0. < N])$.

The proof obligation then, is
$$(K \wedge A[i] = v) \Rightarrow [(i < N) = (v \in A[0. < N])]$$
This is an interesting proof; you should do it. I get the following annotation using $K$ as the invariant

$\{N > 0\}$
$\{0 \leq 0 \leq N \wedge (v \notin A[0. < 0])\}$
    $A[N] := v; \ i := 0;$
$\{0 \leq i \leq N \wedge (v \notin A[0. < i])\}$
     **while** $A[i] \neq v$ **do**
    $\{0 \leq i \leq N \wedge (v \notin A[0. < i]) \wedge A[i] \neq v\}$
    $\{0 \leq i+1 \leq N \wedge (v \notin A[0. < i+1])\}$
     $i := i+1$
    $\{0 \leq i \leq N \wedge (v \notin A[0. < i])\}$
    **od** ;
$\{0 \leq i \leq N \wedge (v \notin A[0. < i]) \wedge A[i] = v\}$
$\{(i < N) = (v \in A[0. < N])\}$
    $found := (i < N)$
$\{found = v \in A[0. < N]\}$

The sad news is that this annotation cannot be proven. We have to show

1. $N > 0 \Rightarrow 0 \leq 0 \leq N \wedge (v \notin A[0. < 0])$

2. $0 \leq i \leq N \wedge (v \notin A[0. < i]) \wedge A[i] \neq v \Rightarrow 0 \leq i+1 \leq N \wedge (v \notin A[0. < i+1])$

3. $0 \leq i \leq N \wedge (v \notin A[0. < i]) \wedge A[i] = v \Rightarrow [(i < N) = (v \in A[0. < N])]$

Proposition (2) cannot be proven. Nowhere have we used the fact that $A[N] = v$. But without this fact we can't even prove termination (and neither can we prove (2)). Let us strengthen the invariant $K$ to $L$ by adding this conjunct
$$L :: \ 0 \leq i \leq N \wedge (v \notin A[0. < i]) \wedge A[N] = v$$
Do another annotation of the program and prove the required propositions. For proof of termination, choose the variant function, $N - i$. Show that,
$$L \Rightarrow (N - i) \geq 0$$
Next, you have to show that each iteration of the loop decreases $(N - i)$, i.e.

$\{L \wedge A[i] = v \wedge (N - i) = t\}$
   $i := i+1$
$\{(N - i) < t\}$

Derive the required propositions to prove this.
Exercise: Try a proof of the above program using the invariant,

$$0 \leq i \leq N \wedge (v \in A[i. < N+1]).$$

## 10.1   Integer Square Root

Given a natural number $k$ it is required to find its integer square root. Specifically, we want to compute a natural number $s$, satisfying $s^2 \le k \wedge (s+1)^2 > k$. In order to get an invariant from this predicate, we replace 1 by a variable, $b$. Also, I add the requirement that $b$ is a power of 2.

$\quad P :: \ s^2 \le k \wedge (s+b)^2 > k \wedge (\exists i : i \ge 0 : b = 2^i)$

The structure of the program is

$\quad \{k \ge 0\}$
$\quad$ initialize;
$\quad \{P\}$
$\quad$ **while** $B$ **do**
$\quad\quad \{P \wedge B\}$
$\quad\quad\quad S$
$\quad\quad \{P\}$
$\quad$ **od**
$\quad \{P \wedge \neg B\} \ \{s^2 \le k \wedge (s+1)^2 > k\}$

It is easy to derive $B$. Looking at the post-condition, we would like $b$ to become 1. Therefore, $B$ is $b \ne 1$. Let us look at $S$ now. The purpose of $S$ is to reduce $b$ (so that $b = 1$ will hold, eventually) while maintaining the invariant. How do we reduce $b$? Since $b$ has to remain a power of 2, a possible program structure is to first reduce $b$ by halving it and then reestablish the invariant. That is, $S$ is the program fragment

$\quad \{P \wedge b \ne 1\}$
$\quad\quad b := b/2; S'$
$\quad \{P\}$

Let us derive the pre-condition for program $S'$. Recall that
$\quad P:: \ [s^2 \le k] \wedge [(s+b)^2 > k] \wedge (\exists i : i \ge 0 : b = 2^i)$

The first conjunct of $P$, $s^2 \le k$, still holds because $s$ has not changed. For the second conjunct of $P$, halving $b$ transforms $(s+b)^2$ to $(s + 2 \times b)^2$; so, we get $(s + 2 \times b)^2 > k$. The last conjunct definitely holds. Thus,

$\quad \{P \wedge b = 1\}$
$\quad\quad b := b/2;$
$\quad \{[s^2 \le k] \wedge [(s + 2 \times b)^2 > k] \wedge (\exists i : i \ge 0 : b = 2^i)\}$

Check this by using the assignment axiom. So, $S'$ has to satisfy,

$\quad \{[s^2 \le k] \wedge [(s + 2 \times b)^2 > k] \wedge (\exists i : i \ge 0 : b = 2^i)\}$
$\quad\quad S'$
$\quad \{[s^2 \le k] \wedge [(s + b)^2 > k] \wedge (\exists i : i \ge 0 : b = 2^i)\}$

Clearly, if $(s + b)^2 > k$ holds before $S'$ we have no work to do (and doing nothing is always safe). But, if $(s + b)^2 \leq k$ we have to reset $s$ or $b$ (or both) to achieve the post-condition. This suggests that we write $S'$ as

$S' ::$ **if** $(s + b)^2 \leq k$ **then** $S''$ **else** **skip** **fi**

We have the pre- and post-conditions of $S'$. So, let us annotate $S'$ fully.

$\{[s^2 \leq k] \wedge [(s + 2 \times b)^2 > k] \wedge (\exists i : i \geq 0 : b = 2^i)\}$
    **if** $(s + b)^2 \leq k$ **then**
    $\{[s^2 \leq k] \wedge [(s + 2 \times b)^2 > k \wedge (s + b)^2 \leq k] \wedge (\exists i : i \geq 0 : b = 2^i)\}$
      $S''$
    $\{[s^2 \leq k] \wedge (s + b)^2 > k \wedge (\exists i : i \geq 0 : b = 2^i)\}$
    **else**
    $\{[s^2 \leq k] \wedge (s + 2 \times b)^2 > k \wedge (\exists i : i \geq 0 : b = 2^i)\}$
    **fi**
$\{[s^2 \leq k] \wedge [(s + b)^2 > k] \wedge (\exists i : i \geq 0 : b = 2^i)\}$

The remaining task is to create $S''$ meeting the given specification. It is easy to see, using the assignment axiom, that $s := s + b$ does the job. Putting the pieces together, we have so far

    initialize;
    **while** $b \neq 1$ **do**
      $b := b/2$;
      **if** $(s + b)^2 \leq k$ **then** $s := s + b$ **else** **skip** **fi**
    **od**

The specification for initialization is,

    $\{k \geq 0\}$
      initialize
    $\{[s^2 \leq k] \wedge (s + b)^2 > k \wedge (\exists i : i \geq 0 : b = 2^i)\}$

We can set $s$ to 0 to meet $s^2 \leq k$. In order to satisfy $[(s + b)^2 > k] \wedge (\exists i : i \geq 0 : b = 2^i)$ one possible assignment is to set $b$ to very high value, say to $2^k$. But that is too inefficient; we have to go through one iteration just to halve $k$ and this will eat up $k$ iterations to bring $b$ down to 1. Another strategy is to start with a small value of $b$ and keep on doubling $b$ until $b^2 > k$. Let us try

    $\{k \geq 0\}$
    $s := 0$;
    **while** $b^2 \leq k$ **do**
      $b := b + b$
    **od**

Exercises:

1. Postulate an invariant for this little program and annotate it.

2. Put all the pieces together in one annotated program.

3. Prove termination of the loops.

# 11 Exercises

1. Design and prove a program that computes the factorial of a natural number.

2. The sequence of Fibonacci numbers are defined by
   $$f_0 = 0, \ f_1 = 1, \ f_{n+2} = f_n + f_{n+1}, \text{ for all } n, n \geq 0$$
   Design and prove a program that computes the sum of the first $N$ Fibonacci numbers in a variable $sum$.

3. The following program divides integer variable $x$ by a positive integer $y$. It stores the quotient in $q$ and the remainder in $r$, both integer variables. Complete the proof.

   $\{y > 0\}$
   $\quad q := 0; \ r := x;$
   $\quad\quad \textbf{while} \ \ y \leq r \ \ \textbf{do}$
   $\quad\quad\quad r := r - y; \ q := q + 1$
   $\quad\quad \textbf{od}$
   $\quad \{q \times y + r = x, r < y\}$

4. The following program – for multiplying the integers $x, y$ and storing the result in $z$ – is a variation of the program given in the notes. It uses a nested loop. Complete the proof.

   $\{x \times y = t, y \geq 0\}$
   $\quad z := 0;$
   $\quad\quad \textbf{while} \ \ y \neq 0 \ \ \textbf{do}$
   $\quad\quad\quad \textbf{while} \ \ even(y) \ \ \textbf{do}$
   $\quad\quad\quad\quad x := 2 \times x; \ y := y \div 2;$
   $\quad\quad\quad \textbf{od} \ ;$
   $\quad\quad\quad z := z + x; \ y := y - 1$
   $\quad\quad \textbf{od}$
   $\quad \{z = t\}$

5. For a positive integer $N$, the following program computes an approximation to the logarithm (base 2) of $N$ in variable $log$. The variable $pow$ below is $2^{log}$.

   $\{N > 0\}$
   $\quad log := 0; \ pow := 1;$
   $\quad\quad \textbf{while} \ \ 2 \times pow \leq N \ \ \textbf{do}$
   $\quad\quad\quad log := log + 1; \ pow := pow \times 2$
   $\quad\quad \textbf{od}$
   $\quad \{2^{log} \leq N < 2^{log+1}\}$

For the following problems, $A[0..N]$ is an array of integers, $N \geq 0$. Develop the programs and proofs together for each of these problems.

6. (a) Find the largest number in $A$ and store it in $hi$.

   (b) Assuming $N \geq 1$, find the largest and the smallest numbers, and store them in $hi, lo$, respectively.

   (c) Solve the problem in (6b) with the following, more sophisticated algorithm. First, compare every adjacent pair of numbers, $A[2 \times i]$ with $A[2 \times i + 1]$, for all $i$, $i \geq 0$ (assume that the length of $A$ is even), storing the smaller one in $A[2 \times i]$ and the larger one in $A[2 \times i + 1]$. Then find the smallest number over all even-indexed elements and store it in $lo$; find the largest one over all odd-indexed elements and store it in $hi$.

   (d) Assuming $N \geq 1$ find the second largest number in $A$ and store it in $sl$.

7. Sort $A$ using the following strategy. Find the smallest number in $A$ and exchange it with $A[0]$. Repeat the process with the array $A[1..N]$.

   Next, consider the following variation. Find the smallest and the largest numbers in $A$, and exchange them with $A[0]$ and $A[N]$, respectively. Repeat the process with the array $A[1..N-1]$.

8. Let $A[0..N]$, $N \geq 0$, and $B[0..M]$, $M \geq 0$, be sorted in ascending order. Merge their elements to form $C[0..M + N + 1]$ in ascending order. (Ascending order for $A$ means $(\forall i : 0 \leq i < N : A[i] \leq A[i + 1])$.)

9. Let $A[0..N]$, $N \geq 0$, and $B[0..M]$, $M \geq 0$, be sorted in ascending order. Detect if they have a common element.

10. Write a program to transpose a matrix and prove its correctness.

11. Write a program to compute the inner product of two vectors, and prove its correctness.

12. You are given a matrix of numbers that has $M$ rows. Write a program that outputs the index of a row that consists of all zeroes. If there is no such row, output $M + 1$.

13. Given a family of sets $in_j, 0 \leq j < N$ and $out_j, 0 \leq j < N$. Set a variable $s$ to *true* iff every set $in_i$ is a subset of every set $out_j$ whenever $i \neq j$. That is,
$$(\forall i, j : 0 \leq i < N \land 0 \leq j < N \land i \neq j : in_i \subseteq out_j).$$

On first glance, a quadratic solution is obvious. We describe a linear solution below.

**Invariant**::
$cin = (\cup j : 0 \le j < k : in_j)$,
$cout = (\cap j : 0 \le j < k : out_j)$,
$s = (\forall i, j : 0 \le i < k \wedge 0 \le j < k \wedge i \ne j : in_i \subseteq out_j)$

**Program**:: Below *ALL* denotes the set of all elements, the universal set.
$cin := \{\}; \; cout := ALL; \; k := 0; \; s := true;$
while $k \ne N$ do
    $s := s \wedge (in_k \subseteq cout) \wedge (cin \subseteq out_k);$
    $cin := cin \cup in_k; \; cout := cout \cap out_k;$
    $k := k + 1$
od

Note: A simpler exercise is: consider arrays $X$ and $Y$ each of whose elements is an integer. Let predicate $s = (\forall i, j : 0 \le i < N \wedge 0 \le j < N \wedge i \ne j : X_i \le Y_j)$. Then $\cup, \cap$ and can be replaced by $min, max$.

14. Take Dijkstra's integer square root problem and solve it using binary search. From $s^2 \le k \; \wedge \; (s+1)^2 > k$, use the invariant, $s^2 \le k \; \wedge \; t^2 > k$ with the termination condition $t = s + 1$.

15. You are given a positive integer $n$. Find all pairs of positive integers $a$ and $b$, $a \le b$, such that $n = a^3 + b^3$. Note that there may be zero, one or more than one of such pairs for a given $n$. For instance, since $1^3 + 1^3 = 2$ and $1^3 + 2^3 = 9$, for all $n$, $2 < n < 9$, there is no such pair. And, 9 has exactly one pair corresponding to it. The smallest positive integer which has two pairs corresponding to it is 1729; $1729 = 10^3 + 9^3$ and $1729 = 12^3 + 1^3$.

    Hint: Imagine a matrix $f$ where $f[i,j] = i^3 + j^3$, $1 \le i \le n^{1/3}/2$, $1 \le j \le n^{1/3}$. Search for $n$ in this matrix using saddleback search.

16. Let $A$ be a 0-1 matrix where the numbers in each column are nondecreasing. Find the first row which has all 1's.

    Hint: Introduce variables $i$ and $j$ which satisfy the invariant

    $$\langle \forall s, t : i \le s < N \; \wedge \; j < t < N : A[s,t] = 1 \rangle$$

    where $A$ is $A[0..N-1, \; 0..N-1]$.

```
i, j = 0, N - 1;
 while  i < N  ∧  j ≥ 0  do
    if  A[i, j] = 1  then  j := j - 1
    else  i := i + 1
    endif
 od ;
```

**if** $i < N$ **then** $\{j < 0\}$ "row $i$ is the desired row"
**else** "there is no such row"
**endif**

17. (Coffee Can Problem) There are $m$ white and $n$ black pebbles in a can, and an infinite supply of black pebbles outside the can. A *move* draws two pebbles from the can; if they are of the same color throw out both pebbles and put back a black pebble in the can, and if they are of different colors, throw out the black pebble and put back the white pebble in the can. The game terminates when there is just one pebble in the can. What is the color of the last pebble?

Note that each move decreases the number of pebbles in the can by one. So, the game definitely terminates. An additional question is: how many additional black pebbles are needed, as a function of $m$ and $n$, to play this game. Assume that if two black pebbles are drawn one of those pebbles is put back, so no additional pebbles are needed.

**Solution**   Each move preserves the parity of the white pebbles. So the last pebble is white iff $m$ is odd.

To calculate the number of additional black pebbles needed, consider three cases. If we draw two whites, we need one black pebble from outside. If we draw two blacks. we don't need any outside black pebble, because we can use on of these. Similarly, with a black and white pebble, we need no external pebble. Therefore, we need one black pebble for each pair of whites we draw (and we lose those two whites in the move). So, we need at most $\lfloor m/2 \rfloor$ external black pebbles.