# *Finding Invariants; Examples*

*CS 536: Science of Programming, Fall 2019*

10/27, 11/10 (typos)

## A. Why

- It is easier to write good programs and check them for defects than to write bad programs and then debug them.
- The hardest part of programming is finding good loop invariants.
- There are heuristics for finding them but no algorithms that work in all cases.
- Changing how we re-establish a loop invariant can greatly speed up the code.

## B. Objectives

At the end of this lecture you should

- Know how to generate possible invariants using the techniques "replace a constant by a variable", "Drop a conjunct" or "Add a disjunct".

## C. Finding Invariants

- The key (and often, hardest) part of writing correct programs involves finding invariants for our loops.
  - We need to find an invariant and loop test that establishes the desired postcondition: $\{$**inv** $p\}$ **while** $B$ **do** ??? **od** $\{p \wedge \neg B\}\ \{r\}$
  - The invariant should be easy to establish with some easy initialization code: $\{p_0\}\ S_0\ \{p\}$.
  - The loop body maintains the invariant: $\{p \wedge B\}$ loop body $\{p\}$
  - When the loop terminates, the postcondition we want holds: $p \wedge \neg B \rightarrow r$. (Sometimes you have finalization code that you need, then you need $\{p \wedge \neg B\}$ *code* $\{r\}$.
- There exist various general heuristics for finding invariants.
  - (Not every way applies to every situation.)
- **General idea**: Take the postcondition and weaken it somehow. The loop test is determined by how and how much you weaken the postcondition.
  - One way to weaken the postcondition: Add more states to it. Possibilities include
    - Adding a new parameter, as in Replace a Constant by a Variable, or Split One Variable Into Two.
    - Making a relation more general. E.g., change an $=$ to a $\leq$ or to an equivalence relation.
    - Add a disjunction (generalize the postcondition $r$ using some $r'$ to get $r \vee r'$ as a possible invariant).
  - Another way to weaken the postcondition: Stop removing states from it.
    - Drop a conjunct (if the postcondition is $p \wedge q$, try using just $p$ or just $q$ for the invariant).

 

### *D. Replace A Constant By A Variable*

- The technique “Replace a constant by a variable” produces a candidate invariant by adding a new parameter to a predicate.
- The idea is to take the postcondition and replace a literal or symbolic constant $c$ with a fresh variable `x`.
- Given the postcondition $r$, find a predicate $r'$, a new variable `x`, and a constant $c$ such that $r'[c/\mathtt{x}] \Leftrightarrow r$.
    - (A generalization is to use any constant-valued subexpression, not just a literal constant.)
    - May want to include the range of `x` as part of $r'$.
    - Our possible loop is $\{$**inv** $r'\}$ **while** $\mathtt{x} \neq c$ **do** … **od** $\{r' \wedge \mathtt{x} = c\}$ $\{r\}$
- Depending on how and what you replace, you get different candidates for invariants, with possibly different loop tests, initialization code, and loop bodies.
- **Example 1**: The summation loops
    - The postcondition $\mathtt{s = sum(0, n)}$ has two constants 0 and `n`.
    - Try replacing `n` by a variable `i` in the range `0`, …, `n`. Initialize `i` = `0` and increase it until `i` = `n`.

        $\{$**inv** $\mathtt{s = sum(0, i)} \wedge 0 \leq \mathtt{i} \leq \mathtt{n}\}$ $\{$**bd** $\mathtt{n-i}\}$

        **while** $\mathtt{i} \neq \mathtt{n}$ **do** … *make* `i` *larger* … **od**

        $\{\mathtt{s = sum(0, i)} \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i = n}\}$

        $\{\mathtt{s = sum(0, n)}\}$
    - Or, replace `0` by a variable `j` in the range `0`, …, `n`. Initialize `j` = `n` and decrease it until `j` = `0`.

        $\{$**inv** $\mathtt{s = sum(j, n)} \wedge 0 \leq \mathtt{j} \leq \mathtt{n}\}$ $\{$**bd** `j`$\}$

        **while** $\mathtt{j} \neq 0$ **do** … *make* `j` *smaller* … **od**

        $\{\mathtt{s = sum(j, n)} \wedge 0 \leq \mathtt{j} \leq \mathtt{n} \wedge \mathtt{j} = 0\}$

        $\{\mathtt{s = sum(0, n)}\}$
- **Example 2**: Integer square root
    - To take the integer square root of an $\mathtt{n} \geq 0$ means to find an `x` such that $\mathtt{x} \leq \mathtt{sqrt(n)} < \mathtt{x+1}$.
    - Let's rewrite the postcondition as $\mathtt{x}^2 \leq \mathtt{n} < (\mathtt{x+1})^2$. We can weaken it by replacing the `1` by a new variable, say `y` and get $\mathtt{x}^2 \leq \mathtt{n} < (\mathtt{x+y})^2$ as a possible invariant. Loop initialization (not shown) sets `y` to something large; the loop body makes `y` smaller.

        $\{$**inv** $\mathtt{x}^2 \leq \mathtt{n} < (\mathtt{x+y})^2 \wedge 1 \leq \mathtt{y}\}$ $\{$**bd** `y`$\}$ // note $\mathtt{y} \geq 1$ can be inferred

        **while** $\mathtt{y} \neq 1$ **do** … *make* `x` *larger or* `x+y` *smaller* … **od** [10/30]

        $\{\mathtt{x}^2 \leq \mathtt{n} < (\mathtt{x+y})^2 \wedge 1 \leq \mathtt{y} \wedge \mathtt{y} = 1\}$

        $\{\mathtt{x}^2 \leq \mathtt{n} < (\mathtt{x+1})^2\}$
- An extended version of the “Replace a constant by a variable” principle is "Replace an expression by a variable". E.g., we might change the variable `x` in `x+1` to `y` and get `y+1` or we could replace the expression `x+1` by `y`, so $\mathtt{x}^2 \leq \mathtt{n} < (\mathtt{x+1})^2$ becomes $\mathtt{x}^2 \leq \mathtt{n} < \mathtt{y}^2$. The loop body either increases `x` or decreases `y`.

    $\{$**inv** $0 \leq \mathtt{x}^2 \leq \mathtt{n} < \mathtt{y}^2\}$ $\{$**bd** `y-x`$\}$

    **while** $\mathtt{y} \neq \mathtt{x+1}$ **do** … *make* `x` *larger or make* `y` *smaller* … **od**

    $\{0 \leq \mathtt{x}^2 \leq \mathtt{n} < \mathtt{y}^2 \wedge \mathtt{y} = \mathtt{x+1}\}$

    $\{0 \leq \mathtt{x}^2 \leq \mathtt{n} < (\mathtt{x+1})^2\}$

 

- For termination, $0 \le x^2 < y^2$ implies $y \ge x+1$, so $y-x \ge 0$, and reducing $y$ or increasing $x$ reduces $y-x$.

### *Loop Initialization When Replacing a Constant by a Variable*

- For loop initialization, we typically establish the invariant by setting variables to some boundary values.
  - E.g., if $c_0 \le v \le c_1$, try $v := c_0$ or $v := c_1$ as initializations.
- **Example 3**: Summation loops
  - For the invariant $s = sum(0, i) \wedge 0 \le i \le n$, setting $i := 0$ or $i := n$ seems natural:
  - $wp(i := 0, p) \equiv s = sum(0, 0) \wedge 0 \le 0 \le n$ is easy to establish with $s := 0$ (and the assumption $n \ge 0$).
  - But $wp(i := n, p) \equiv s = sum(0, n) \wedge 0 \le n \le n$ is hard to satisfy (in fact, it's our original postcondition).
- **Example 4**: For $x^2 \le n < y^2 \wedge x < y$, try $x := 0$ or $x := 1$ (these imply we need $0^2 \le n$ or $1^2 \le n$ respectively). For $y$, we can try $y := n$ (if we know $n > 1$, so that $n < n^2$) or $y := n+1$ (if we know only $n \ge 1$) or $y := n+2$ (if we know only $n \ge 0$).

### *Ensuring Loop Termination When Replacing a Constant by a Variable*

- A loop always has to include at least one **progress statement**; a statement that gets us closer to termination. If a progress statement $S_2$ is put at the end of the loop body, then the rest of the loop body $S_1$ has to satisfy

$$\{p \wedge B \wedge t = t_0\}\ S_1\ \{wp(S_2, p \wedge t < t_0)\}$$

- So as our loop, we have $\{\mathbf{inv}\ p\}\ \{\mathbf{bd}\ t\}\ \mathbf{while}\ B\ \{p \wedge B \wedge t = t_0\}\ S_1\ \mathbf{;}\ S_2\ \{p \wedge t < t_0\}\ \mathbf{od}$.
- When replacing a constant by a variable, the progress statement takes the variable closer to the target constant.

### *Two Simple Assignments for Establishing the Value of a Variable*

- Say we want $S$ such that $\{v = e_1\}\ S\ \{v = e_2\}$. Two simple ways are:
  - $\{v = e_1\}\ v := v + e_2 - e_1\ \{v = e_2\}$
  - $\{v = e_1\}\ v := v * e_2 \div e_1\ \{v = e_2\}$ // (assuming $e_1$ divides $e_2$)
- One example was in the summation loop: We needed $s = sum(0, i+1)$ but had $s = sum(0, i)$. We use

  $\{s = sum(0, i)\}\ s := s + (i+1)\ \{s = sum(0, i+1)\}$

  because it is equivalent to (the harder-to-calculate)

  $\{s = sum(0, i)\}\ s := s + sum(0, i+1) - sum(0, i)\ \{s = sum(0, i+1)\}$
- **Example 5**: Find the largest power of 2 that is $\le x$.
  - Say our invariant is $y = 2^k \le x \wedge 0 \le k$ (we loop **while** $2*y \le x$) and our progress step is $k := k+1$, so the *wp* of the progress step is $y = 2^{k+1} \le x \wedge 0 \le k+1$.
  - So we need code to establish $\{y = 2^k \wedge \ldots\}$; $y :=$ ??? $\{y = 2^{k+1} \wedge \ldots\}\ k := k+1\ \{y = 2^k \wedge \ldots\}$
    - One possibility is $y := y + 2^{k+1} - 2^k$. I.e., $y := y + 2^k$, or just $y := y + y$, since $y = 2^k$.
    - Another possibility for our statement is $y := y * 2^{k+1} \div 2^k$, which simplifies to $y := y * 2$.

### *Replacing a Constant by a Variable Can Fail*

- Not every constant when replaced yields an invariant that works well.

- E.g. take the postcondition $x^2 \le n < (x+1)^2$ and replace one (or say both) of the 2's with a new variable y. We loop **while** $y \ne 2$ with a proposed invariant of
  - $x^2 \le n < (x+1)^y$ plus something for the range of y.
  - or $x^y \le n < (x+1)^2$ plus something for the range of y.
- How would we initialize y? If we're using $x^y \le n$ we could try $y := 0$ so we'd need $1 = x^0 \le n$. Less obvious what to use if we're trying $n < (x+1)^y$. Maybe $x := n$; $y := 1$ ? But we'd need $n^2 \le n < (n+1)^1$, and $n^2 \le n$ requires $n = 0$ or 1, which seems kind of limiting.
- Progress step: If $y := y+1$ (for example) is the progress step, then the rest of the loop body needs to be the missing code in

  $\{x^y \le n < (x+1)^2 \wedge y \ne 2\}$ … $\{x^{y+1} \le n < (x+1)^2\}$ $y := y+1$ $\{x^y \le n < (x+1)^2\}$
  - What could the missing code possibly be? Time to give up and look for a different invariant.

### *E. Deleting A Conjunct*

- Deleting a conjunct is another way to find possible invariants. To use it, we need a postcondition that is the conjunction of multiple conjuncts. Say postcondition is $r$ is $p_1 \wedge p_2 \ldots \wedge p_n$ where $n \ge 2$.
  - Let, $Less(r, k) \equiv (p_1 \wedge p_2 \ldots \wedge p_{k-1}) \wedge (p_{k+1} \wedge \ldots \wedge p_n)$. I.e., $r$ "less" the conjunct $p_k$.
- There are $n$ possible invariants, one for each conjunct. In general, for conjunct $k$ we have

  {**inv** $p \equiv Less(r, k)$}
  **while** $\neg p_k$ **do**
  $\quad\{p \wedge \neg p_k\}$ … $\{p\}$
  **od**
  $\{p \wedge p_k\}$ $\{r\}$

### *Example 6: Linear Search of an Array*

- Precondition: Array b has at least n elements ($n \ge 0$) and the value x may or may not appear in b[0..n−1].
- Postcondition: We find the index k of the leftmost occurrence of x in b[0..n−1]. If x doesn't appear in b[0..n−1], then $k = n$. Note in either case, x doesn't appear in b[0..k−1]. We can formalize this as

  $$0 \le k \le n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)$$

  where $x \notin b[0..k-1]$ means $\forall\, 0 \le k' < k\,.\, x \ne b[k']$. Note if $k = 0$, then $b[0..k-1] = b[0..-1]$ is the empty sequence of values.
- Since $0 \le k \le n$ is short for $0 \le k \wedge k \le n$, there are four conjuncts we can try deleting, which yields four possible loop/test combinations. Three of them don't yield a usable invariant, but the fourth one does.
  - {**inv** $k \le n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)$} // Drop the conjunct $0 \le k$
    **while** $0 > k$ **do** …

    If we use this, then in the loop body we have $k < 0$, which makes referencing b[k] illegal. This sounds really unpromising.

- {**inv** $0 \le k \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)$} // Drop the conjunct $k \le n$
  **while** $k > n$ **do** …

  This has the symmetric problem: k is too large to be an index.

- {**inv** $0 \le k \le n \wedge (k < n \rightarrow b[k] = x)$} // Drop the conjunct $x \notin b[0..k-1]$
  **while** $x \in b[0..k-1]$ **do** …

  There are two problems with this proposed invariant. First, how do we initialize k? Setting k := 1 would require b[0] = x, and setting k := n requires knowing that x doesn't appear in b[0..n-1]. The second problem is that the test $x \in b[0..k-1]$ takes time proportional to k, since we'll need a loop or recursion to write it.

- The fourth possibility, however, works well. Here, we drop $k < n \rightarrow b[k] = x$.

  {**inv** $0 \le k \le n \wedge x \notin b[0..k-1]$}
  **while** $\neg(k < n \rightarrow b[k] = x)$ **do** …

- Let's borrow the short-circuiting && operator from C: if $e_1$ and $e_2$ are boolean expressions then

  $e_1$ && $e_2 \equiv$ **if** $e_1$ **then** $e_2$ **else F fi**.

- Now we can rewrite $\neg(k < n \rightarrow b[k] = x)$ as $k < n$ && $b[k] \ne x$.
- Initialization is easy: k := 0, since its *wp* is $0 \le 0 \le n \wedge x \notin b[0..0-1]$. The only nontrivial part is $n \ge 0$, which will be the initial precondition.
- Since k starts out at 0 and must increase to n, a progress step of k := k+1 seems pretty reasonable. The loop body so far is

  {$p \wedge k < n \wedge b[k] \ne x$} // Invariant ∧ loop test
  ???
  {$0 \le k+1 \le n \wedge x \notin b[0..k+1-1]$} // *wp* of progress step
  k := k+1 // Progress step
  {$0 \le k \le n \wedge x \notin b[0..k-1]$} // Invariant

  where ??? will be code that can take us from the precondition of the loop body (*invariant* ∧ *test*) to the *wp* of the loop body (i.e., *wp*(*progress step*, *invariant*)). But it turns out that we don't need any code to do this.

- Convergence is easy: Since *p* includes $k \le n$ and k gets incremented, we can use n−k. So the whole loop is

  {$n \ge 0$} k := 0;
  {**inv** $p \equiv 0 \le k \le n \wedge x \notin b[0..k-1]$} {**bd** n−k}
  **while** $k < n$ && $b[k] \ne x$ **do**
  {$p \wedge n-k \ge 0 \wedge k < n \wedge b[k] \ne x \wedge n-k = t_0$} // made p → n-k ≥ 0 more explicit. [10/30]
  {$0 \le k+1 \le n \wedge x \notin b[0..k+1-1] \wedge n-(k+1) < t_0$}
  k := k+1
  {$p \wedge n-k < t_0$}
  **od** {$0 \le k \le n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)$}

### *F. Adding a Disjunct*

- Adding a disjunct is another way to find possible invariants. Say we want to establish postcondition $r$. For various possible $B$, we can try

  $\{$**inv** $r \vee B\}$
  **while** $B$ **do**
  $\{(r \vee B) \wedge B\}$ Loop body $\{r \vee B\}$
  **od** $\{(r \vee B) \wedge \neg B\}$ $\{r\}$

- Unlike first two methods, this one is very open-ended — you can use any testable predicate for $B$.
- Adding a disjunct lets us, e.g., generalize a relation like $\mathtt{i} = \mathtt{n}$ to $\mathtt{i} \le \mathtt{n}$ (i.e., $\mathtt{i} = \mathtt{n} \vee \mathtt{i} < \mathtt{n}$). This is one way to understand a loop like $\{$**inv** $\mathtt{i} \le \mathtt{n}$ … $\}$ **while** $\mathtt{i} < \mathtt{n}$ **do** … **od** $\{\mathtt{i} = \mathtt{n}\}$: The postcondition $\mathtt{i} = \mathtt{n}$ gets the disjunct $\mathtt{i} < \mathtt{n}$ added and becomes $\mathtt{i} \le \mathtt{n}$ in the invariant.
- Adding a disjunct is one way to view deleting a conjunct: Changing $p \wedge q$ to $(p \wedge q) \vee (p \wedge \neg q)$ yields something $\Leftrightarrow$ just $p$.
- Converting $p \wedge q$ to $p \vee q$ can be viewed as a generalization of $\wedge$ to $\vee$ or as taking $p \wedge q$ to $(p \wedge q) \vee (p \wedge \neg q) \vee q$.

------------------------------------------------------------------------------- ended 10/30

### *G. Example 7: Binary Search Example (Version 1)*

- Binary search is a nice example of a loop that isn't a **for** loop. For termination, a loose upper bound (the distance between the endpoints) suffices.
- Program specification: $\{q_0\}$ *Binsearch*$(\mathtt{b}, \mathtt{x}, \mathtt{n})$ $\{r\}$ where
  - $q_0 \equiv \mathtt{Sorted(b, n)} \wedge 1 \le \mathtt{n} < \mathtt{size(b)} \wedge \mathtt{b[0]} \le \mathtt{x} < \mathtt{b[n]}$
  - $\mathtt{Sorted(b, n)} \equiv \forall\, 0 \le \mathtt{i} < \mathtt{n-1} < \mathtt{size(b)-1}\,.\ \mathtt{b[i]} \le \mathtt{b[i+1]}$.
  - $r \equiv 0 \le \mathtt{L} < \mathtt{n} \wedge (\mathtt{found} \leftrightarrow \mathtt{x} = \mathtt{b[L]})$
- Having $\mathtt{x} < \mathtt{b[n]}$ means $\mathtt{b[n]}$ is a sentinel value, not an actual data value.
- Let's treat $\mathtt{b}$ and $\mathtt{n}$ as named constants so that $\mathtt{Sorted(b, n)}$ can be used anywhere and doesn't have to be part of the invariant.
- For our invariant, we can generalize the initial precondition $\mathtt{b[0]} \le \mathtt{x} < \mathtt{b[n]}$ to $\mathtt{b[L]} \le \mathtt{x} < \mathtt{b[R]}$ where $0 \le \mathtt{L} < \mathtt{R} \le \mathtt{n}$. In addition, we can weaken the postcondition's $(\mathtt{found} \leftrightarrow \mathtt{x} = \mathtt{b[L]})$ to just implication: $(\mathtt{found} \rightarrow \mathtt{x} = \mathtt{b[L]})$; this lets us have $\mathtt{found} = \mathtt{F}$ while we search. For the search bound, we can use $\mathtt{R-L}$; it's a loose termination bound but that's okay.
- For the loop body, we'll begin by calculating the midpoint $\mathtt{m} := \mathtt{(L+R)/2}$ (with truncating division). Clearly, if $\mathtt{b[m]} = \mathtt{x}$, we can set $\mathtt{found}$ to true and $\mathtt{L}$ to $\mathtt{m}$ and exit the loop.
- The loop so far is

  $\{q \equiv \mathtt{Sorted(b, n)} \wedge \mathtt{n} \ge 1 \wedge \mathtt{b[0]} \le \mathtt{x} < \mathtt{b[n]}\}$
  $\mathtt{L := 0\ ;\ R := n\ ;\ found := F\ ;}$
  $\{$**inv** $p \equiv 0 \le \mathtt{L} < \mathtt{R} \le \mathtt{n} \wedge \mathtt{b[L]} \le \mathtt{x} < \mathtt{b[R]} \wedge (\mathtt{found} \rightarrow \mathtt{x} = \mathtt{b[L]})\}$ $\{$**bd** $\mathtt{R-L}\}$
  **while** $\neg\mathtt{found} \wedge \mathtt{R} \ne \mathtt{L+1}$ **do**
  $\{p \wedge \neg\mathtt{found} \wedge \mathtt{R} \ne \mathtt{L+1} \wedge \mathtt{R-L} = \mathtt{t_0}\}$

```
        m := (L+R)/2 ;
        {p ∧ ¬found ∧ R ≠ L+1 ∧ R-L = t₀ ∧ m = (L+R)/2}
        if b[m] = x then
            found := T ; L := m ; R := L+1 [11/4] (add b[L} ≤ x < b[L+1] at end?)
        else
            // … to be filled in …
        fi
        {p ∧ R-L < t₀}
od
{p ∧ (found ∨ R = L+1)}
{0 ≤ L < n ∧ (found ↔ x = b[L])}
```

- It's easy to verify that loop initialization is correct. At loop termination, either found is true and $b[L] = x$, or found is false, $R = L+1$, and $b[L] < x < b[L+1]$, indicating the search has indeed failed.
- If $b[m] \neq x$, we make progress toward termination by setting L or R to m. To reestablish the invariant, we need

  $\{p[m/L] \wedge R-m < t_0\}$ L := m $\{p \wedge R-L < t_0\}$

  or $\{p[m/R] \wedge m-L < t_0\}$ R := m $\{p \wedge R-L < t_0\}$

  - In the first case, we need $0 \leq m < R \leq n \wedge b[m] \leq x < b[R] \wedge (found \rightarrow x = b[m] \wedge R-m < t_0)$.
  - In the second case, we need $0 \leq L < m \leq n \wedge b[L] \leq x < b[m] \wedge (found \rightarrow x = b[L] \wedge m-L < t_0)$.
  - We already know $b[m] \neq x$, so testing $b[m] < x$ vs $b[m] > x$ will establish which of these two cases we are in. We also need $R-m < t_0$ or $m-L < t_0$, where $t_0 = R-L$; these both follow from $L < m < R$, which in turn follows from $L < R \wedge R \neq L+1$. (Since $L+2 \leq R$, $m = (L+R)/2$ is $\geq (2*L+2)/2 = L+1$ and also $\leq (2*R-2)/2 < R$.)
- This gives us a loop body partially outlined as

```
{p ∧ ¬found ∧ R ≠ L+1 ∧ R-L = t₀}
m := (L+R)/2 ;
{p₁ ≡ p ∧ ¬found ∧ R ≠ L+1 ∧ R-L = t₀ ∧ m = (L+R)/2}
if b[m] = x then
    found := T ; L := m
else if b[m] < x then
    L := m
else // b[m] > x
    R := m
fi fi
{p ∧ R-L < t₀}
```

- (One of the activity questions is to fill out the annotation.)

## *H. Example 8: Traditional Binary Search*

- For contrast, let's look at a traditional version of binary search, where we stop if $L > R$.

- We begin with almost the same precondition, $\text{Sorted}(b, n) \wedge n \geq 1 \wedge b[0] \leq x \leq b[n]$. (We weakened $x < b[n]$ to $x \leq b[n]$.)
- The postcondition will be different: If we end with $R < L$ (in particular $R = L-1$) then the search has failed, otherwise $b[L] = x$ as before. Again, to distinguish between failure and success, we'll use found to stop the search. At termination,

  $$-1 \leq L-1 \leq R < n \wedge (\text{found} \rightarrow b[L] = x) \wedge (\neg\text{found} \rightarrow x \notin b[0..n-1])$$

  - (The first conjunct, $-1 \leq L-1 \leq R < n$, summarizes the properties and relationships of L and R, namely $0 \leq L < n$ and either $L \leq R < n$ or $R = L-1$.)
- For the invariant, we want to weaken $(\neg\text{found} \rightarrow x \notin b[0..n-1])$ to something that will be true during the search. I'll use $(x \in b[0..n-1] \leftrightarrow x \in b[L..R])$ with the understanding that when $R = L-1$ then $b[L..R] = b[L..L-1] = \emptyset$. This way, if $R < L$, we know the search has failed. We should terminate the loop if found or ($R < L$ [and $\neg$found]).
- Now for a bound function. We can't use $R-L$ because it can be $-1$. We can almost use $R-L+1$, except that when find $b[m] = x$, all we do is set found := true and L := m, which doesn't necessarily decrease $R-L+1$. To take found into account, define $|F| = 0$ and $|T| = 1$, then we can use $R-L+1+|\neg\text{found}|$ for the bound function.
- Altogether, we get the following sketch for our binary search:

```
{n > 0 ∧ Sorted(b, n) ∧ b[0] ≤ x ≤ b[n-1]}
L := 0; R := n-1; found := F;
{inv q ≡ -1 ≤ L-1 ≤ R < n ∧ (found → b[L] = x) ∧ (x in b[0..n-1] ↔ x in b[L..R])}
{bd R-L+1+|¬found|}
while ¬found ∧ L ≤ R do
      m := (L+R)/2;
      {q1 ≡ q ∧ ¬found ∧ L ≤ R ∧ R-L+1+|¬found| = t0 ∧ m = (L+R)/2}
      if b[m] = x then
            found := T ; L := m
      else if b[m] < x then
            L := m+1
      else // b[m] > x
            R := m-1
      fi fi
od
{q ∧ (found ∨ L > R)}
{-1 ≤ L-1 ≤ R < n ∧ (found → b[L] = x) ∧ (¬found → x ∉ b[0..n-1])}
```

### *Example 9: Match across two lists*

- We have two sorted arrays $b_1$ and $b_2$ and want to find the least indexes i and j that make $b_1[i] = b_2[j]$; if no such values exist, we should halt with $i = n \vee j = m$.

- We'll use a bound function of $(n-i) + (m-j)$. We can initialize i and j to 0, increment at least one of them with each iteration and ensure that the invariant implies $0 \le i \le n \wedge 0 \le j \le m$.

- We aren't going to change $b_1$ or $b_2$, so we can specify $\text{Sorted}(b_1, n) \wedge \text{Sorted}(b_2, m)$ in the initial precondition, but after that we can omit it as being implicit.

$$\text{Sorted}(b, n) \equiv \forall\, 0 \le k \le n-2 \,.\, b[k] \le b[k+1]$$

- We can formalize the "least indexes i and j" part of the postcondition as a property that says no value to the left of $b_1[i]$ matches any value to the left of $b_2[j]$:

$$\text{NoMatch}(i, j) \equiv \forall\, 0 \le i' < i \le n \,.\, \forall\, 0 \le j' < j \le m \,.\, b_1[i'] \ne b_2[j']$$

- Also, let $\text{InRange}(i, j) \equiv 0 \le i \le n \wedge 0 \le j \le m$, then our postcondition is

$$q \equiv \text{InRange}(i, j) \wedge \text{NoMatch}(i, j) \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$$

- To get an invariant, we'll drop the third conjunct (or add a disjunct of $(i < n \wedge j < m \rightarrow b_1[i] \ne b_2[j])$):

  $\{\textbf{inv}\ p \equiv \text{InRange}(i, j) \wedge \text{NoMatch}(i, j)\}$

  $\textbf{while}\ \neg(i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\ \textbf{do} \ldots \textbf{od}$

  $\{p \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\}\ \{q\}$

As in linear search (Example 6), we'll rewrite the test as $B \equiv (i < n \wedge j < m \ \&\&\ b_1[i] \ne b_2[j])$. As a conditional expression, this is $\textbf{if}\ i < n \wedge j < m\ \textbf{then}\ b_1[i] \ne b_2[j]\ \textbf{else}\ F\ \textbf{fi}$.

- Before writing the loop body, let's consider initialization. As we begin, NoMatch(0, 0) is all we know about the arrays, we can set i and j to zero.

  $\{n \ge 0 \wedge m \ge 0 \wedge \text{Sorted}(b, n) \wedge \text{Sorted}(b_2, m)\}$

  $i := 0;\ j := 0\ \{\text{Range}(0, 0) \wedge \text{NoMatch}(0, 0)\}$

  $\{\textbf{inv}\ p \equiv \text{InRange}(i, j) \wedge \text{NoMatch}(i, j)\}\ \{\textbf{bd}\ (n-i) + (m-j)\}$

  $\textbf{while}\ i < n \wedge j < m \ \&\&\ b_1[i] \ne b_2[j]\ \textbf{do} \ldots \textbf{od}$

  $\{q \equiv p \wedge B\}$ // where $B \equiv i < n \wedge j < m \rightarrow b_1[i] = b_2[j]$ [11/4]

- For termination, we need the invariant to imply $(n-i) + (m-j) \ge 0$, which follows from InRange(i, j).

- To get closer to termination, either $i := i+1$ or $j := j+1$ will do. So our loop body will include finding code taking us from the invariant and loop test to the *wp* of each progress statement

  - $\{p \wedge \neg B\}$ ??? $\{\text{InRange}(i+1, j) \wedge \text{NoMatch}(i+1, j)\}\ i := i+1\ \{p\}$
  - $\{p \wedge \neg B\}$ ??? $\{\text{InRange}(i, j+1) \wedge \text{NoMatch}(i, j+1)\}\ j := j+1\ \{p\}$
  - (Recall $p \equiv \text{InRange}(i, j) \wedge \text{NoMatch}(i, j)$ and $\neg B \Leftrightarrow i < n \wedge j < m \ \&\&\ b_1[i] \ne b_2[j]$.)

- $\text{InRange}(i, j) \wedge \ldots i < n \wedge j < m \ldots$ implies InRange(i+1, j) and InRange(i, j+1).

- So the question for $i := i+1$ is how to get from $\text{NoMatch}(i, j) \wedge b_1[i] \ne b_2[j]$ to NoMatch(i+1, j)? Some logic tells us that if we assume $p \wedge \neg B$, then $b_1[i] > b_2[j]$ will ensure NoMatch(i+1, j) because the elements $b_2[j], b_2[j-1], \ldots b_2[0]$ are nondecreasing and the loop test included $b_1[i] \ne b_2[j]$.

  - Altogether, we get $\{p \wedge \neg B\}\ \textbf{if}\ b_1[j] > b_2[i] \rightarrow \{p[i+1/i]\}\ i := i+1\ \textbf{fi}\ \{p\}$ as one (nondeterministic) case for the loop body. [11/4 check tests to be sure]

- Symmetrically, $b_2[j] > b_1[i]$ will ensure NoMatch(i, j+1). This gives us another loop body case:

  $\{p \wedge \neg B\}\ \textbf{if}\ b_2[j] > b_1[i] \rightarrow \{p[j+1/j]\}\ j := j+1\ \textbf{fi}\ \{p\}$

- If we combining these two cases with nondeterministic **if-fi**, we get the (pleasingly?) symmetric

  $\{p \wedge \neg B\}$
  **if** $\texttt{b}_1\texttt{[i]} > \texttt{b}_2\texttt{[j]} \rightarrow \{p[\texttt{i+1/i}]\}\ \texttt{i := i+1}$
  □ $\texttt{b}_2\texttt{[j]} > \texttt{b}_1\texttt{[i]} \rightarrow \{p[\texttt{j+1/j}]\}\ \texttt{j := j+1}$
  **fi** $\{p\}$

- Since the loop test implies $\texttt{b}_1\texttt{[i]} \neq \texttt{b}_2\texttt{[j]}$, we've covered all the possible cases and also ensured that the **if-fi** won't cause a domain error (where none of the tests hold). This means the nondeterministic **if-fi** above can be used as the loop body. To rewrite the **if-fi** deterministically, since we know $\texttt{b}_1\texttt{[i]} \neq \texttt{b}_2\texttt{[j]}$, if $\texttt{b}_1\texttt{[i]} > \texttt{b}_2\texttt{[j]}$ is false, then $\texttt{b}_2\texttt{[j]} > \texttt{b}_1\texttt{[i]}$ must hold. This gives us

  $\{p \wedge \neg B\}$ **if** $\texttt{b}_1\texttt{[i]} > \texttt{b}_2\texttt{[j]}$ **then** $\texttt{i := i+1}$ **else** $\texttt{j := j+1}$ **fi** $\{p\}$

- Adding this to the loop framework (initialization and test), we get [11/4 make full outline for total correct.]

  $\{\texttt{n} \geq 0 \wedge \texttt{m} \geq 0 \wedge \texttt{Sorted(b, n)} \wedge \texttt{Sorted(b}_2\texttt{, m)}\}$
  $\texttt{i := 0; j := 0}$
  $\{$**inv** $p \equiv \texttt{InRange(i, j)} \wedge \texttt{NoMatch(i, j)}\}$ $\{$**bd** $\texttt{n-i + m-j}\}$
  **while** $\neg B$ **do** $\{p \wedge \neg B\}$ // $\neg B \Leftrightarrow \texttt{i < n} \wedge \texttt{j < m \&\& b}_1\texttt{[i]} \neq \texttt{b}_2\texttt{[j]}$
      **if** $\texttt{b}_1\texttt{[i]} > \texttt{b}_2\texttt{[j]}$ **then**
          $\{p \wedge \neg B \wedge \texttt{b}_1\texttt{[i]} > \texttt{b}_2\texttt{[j]}\}\ \texttt{i := i+1}\ \{p\}$
      **else**
          $\{p \wedge \neg B \wedge \texttt{b}_1\texttt{[i]} < \texttt{b}_2\texttt{[j]}\}\ \texttt{j := j+1}\ \{p\}$
      **fi**
  **od** $\{p \wedge B\}$ $\{p \wedge (\texttt{i < n} \wedge \texttt{j < m} \rightarrow \texttt{b}_1\texttt{[i]} \neq \texttt{b}_2\texttt{[j]})\}$

- One interesting property of the nondeterministic solution is that it's easily extendable to more than two arrays. We can add a third array, $\texttt{b}_3$ with index $\texttt{k}$ and size $\texttt{p}$.
  - The invariant becomes $\texttt{i < n} \wedge \texttt{j < m} \wedge \texttt{k < p} \rightarrow \texttt{b}_1\texttt{[i]} \neq \texttt{b}_2\texttt{[j]} \vee \texttt{b}_2\texttt{[j]} \neq \texttt{b}_3\texttt{[k]}$

  $\{p \wedge \neg B\}$
  **if** $\texttt{b}_1\texttt{[i]} > \texttt{b}_2\texttt{[j]} \rightarrow \{p[\texttt{i+1/i}]\}\ \texttt{i := i+1}$ [11/4 … > $b_2$[k]?]
  □ $\texttt{b}_2\texttt{[j]} > \texttt{b}_1\texttt{[i]} \rightarrow \{p[\texttt{j+1/j}]\}\ \texttt{j := j+1}$
  □ $\texttt{b}_3\texttt{[k]} > \texttt{b}_2\texttt{[j]} \rightarrow \{p[\texttt{k+1/k}]\}\ \texttt{k := k+1}$
  **fi** $\{p\}$

## I. *Example 10: Multiply Integers x and y (version 1: Slowly)*

- Our specification is $\{\texttt{x} = \texttt{x}_0 \wedge \texttt{y} = \texttt{y}_0\}\ S\ \{\texttt{z} = \texttt{x}_0\texttt{*y}_0\}$. ($\texttt{x}_0$ and $\texttt{y}_0$ are the initial values of $\texttt{x}$ and $\texttt{y}$.)
  - When the loop ends, we want $\texttt{z} = \texttt{x}_0\texttt{*y}_0$.
  - When the loop begins, we have $\texttt{x}_0\texttt{*y}_0 = \texttt{x*y}$ because $\texttt{x} = \texttt{x}_0 \wedge \texttt{y} = \texttt{y}_0$.
- To get an invariant, we can reframe the definition of $\texttt{z}$ so that it covers both cases: $\texttt{z} = \texttt{x}_0\texttt{*y}_0 - \texttt{x*y}$.
  - When the loop begins, $\texttt{x} = \texttt{x}_0$ and $\texttt{y} = \texttt{y}_0$, so $\texttt{x}_0\texttt{*y}_0 = \texttt{x*y}$, so we'll set $\texttt{z := 0}$.
  - We can end the loop if $\texttt{x}$ or $\texttt{y} = 0$, because $\texttt{z} = \texttt{x}_0\texttt{*y}_0 - \texttt{x*y} = \texttt{x}_0\texttt{*y}_0 - 0$.

    - If $x_0 \ge 0$ initially, then we can maintain $0 \le x \le x_0$, and we make progress by moving x from $x_0$ toward 0. Let's use x := x−1 as the progress step toward termination.
- Combining everything so far with $x \ne 0$ as the loop test gives us

  $\{x = x_0 \ge 0 \wedge y = y_0\}$ z := 0;
  $\{$**inv** $p \equiv x \ge 0 \wedge z = x_0*y_0 - x*y\}$ $\{$**bd** x$\}$
  **while** $x \ne 0$
  **do**
  $\qquad\{p \wedge x \ne 0\}$ *code to write* **;**
  $\qquad\{w\}$ x := x−1 $\{p\}$ // where $w \equiv wp($x := x−1$, p)$
  **od**
  $\{p \wedge x = 0\}\{z = x_0*y_0\}$
- Above, $w \equiv wp($x := x−1$, p) \equiv p[x-1/x] \equiv (z = x_0*y_0 - (x-1)*y \wedge x-1 \ge 0)$
- The loop body precondition $p \wedge x \ne 0 \equiv (z = x_0*y_0 - x*y \wedge x \ge 0) \wedge x \ne 0$
- Note $p$ implies $z = x_0*y_0 - x*y$, but $w$ requires $z = x_0*y_0 - (x-1)*y$.
    - So we don't have $p \wedge x \ne 0 \rightarrow w$, so we need some code between them to establish this.
    - Recall one way to change $z = e_1$ to $z = e_2$ is z := z + $(e_2 - e_1)$. Here, $e_2 - e_1$ is $(x_0*y_0 - x*y) - (x_0*y_0 - (x-1)*y) = x*y - (x-1)*y = y$
    - So $\{p \wedge x \ne 0\}$ z := z+y $\{w\}$ x := x−1 $\{p\}$
- Our program is

  $\{x = x_0 \ge 0 \wedge y = y_0\}$ z := 0;
  $\{$**inv** $p \equiv z = x_0*y_0 - x*y \wedge x \ge 0\}$ $\{$**bd** x$\}$
  **while** $x \ne 0$ **do**
  $\qquad\{p \wedge x \ne 0 \wedge x = t_0\}$ $\{p[x-1/x][z+y/z] \wedge x-1 < t_0\}$
  $\qquad$z := z+y; $\{p[x-1/x] \wedge x-1 < t_0\}$
  $\qquad$x := x−1 $\{p \wedge x < t_0\}$
  **od**
  $\{p \wedge x = 0\}\{z = x_0*y_0\}$
- Partial correctness of this outline is easy to verify. For total correctness, we need to make sure x can be a bound expression.
- The invariant contains $x \ge 0$ as a conjunct, so *invariant* $\rightarrow$ *bound* $\ge 0$ holds.
- The loop body decrements x, so $\{$*invariant* $\wedge$ *loop test* $\wedge$ *bound* $= t_0\}$ *loop body* $\{$*bound exp* $< t_0\}$ holds.

## *J. Example 11: Multiply Integers x and y (version 2: More Quickly)*

### *Progress Step Governs Runtime*

- The program just finished to multiply integers has a runtime linear in $x_0$. We can get a faster multiplication program if we make progress toward $x = 0$ more quickly.
- What if we try x := x÷2?

- We can still use $\texttt{x}$ as the bound expression: The invariant still implies $\texttt{x} \geq 0$, and if $\texttt{x} \neq 0$, then $\texttt{x := x÷2}$ brings us strictly closer to 0.
- Instead of a loop body of

  $\{p \wedge x \neq 0 \wedge x = t_0\}$ `z := z+y; x := x-1` $\{p \wedge x < t_0\}$

  we have

  $\{p \wedge x \neq 0 \wedge x = t_0\}$ ??? $\{w_1\}$ `x := x÷2` $\{p \wedge x < t_0\}$

  where $w_1 \equiv wp(\texttt{x := x÷2}, p \wedge x < t_0)$
  $\equiv (p \wedge x < t_0)[x÷2/x]$
  $\equiv p[x÷2/x] \wedge x÷2 < t_0$
  $\equiv (z = x_0 * y_0 - (x÷2)*y) \wedge x÷2 \geq 0 \wedge x÷2 < t_0$
- The missing statement has to take us from $p \wedge x \neq 0 \wedge x = t_0$ to $w_1$.
  - We're already ensured that the $x÷2 \geq 0$ and $x÷2 < t_0$ clauses of $w_1$ hold:
    - $p$ implies $x \geq 0$, so we know $x÷2 \geq 0$.
    - $x = t_0$ and $x \geq 0 \wedge x \neq 0$ implies $x÷2 < t_0$.
- We need code to go from $(z = x_0 * y_0 - x*y)$ in $p$ to $(z = x_0 * y_0 - (x÷2)*y)$ in $w_1$.
  - If $\texttt{x}$ is even, then $(x÷2)*(2*y) = x*y$.
    - So $\{p \wedge even(x)\}$ `y := 2*y;` $\{w_1\}$ `x := x÷2` $\{p\}$
- But we don't know that $\texttt{x}$ is even. We could check for it:

  **if** even(x)
  **then** … *code* above (*requires* x *to be even*) … $\{w_1\}$
  **else**
  $\{p \wedge x \neq 0 \wedge odd(x)\}$ ??? $\{w_1\}$
  **fi**
- Or we could **force** x to be even:

  $\{p\}$ **if** odd(x) **then** ??? **;** `x := x-1` **fi;** $\{p \wedge even(x)\}$
  … *above code* … $\{w_1\}$
- But **we already know what we can use** before the decrement of x.
  - We've already written it once: it's `z := z+y`.
- This completes the program:

  $\{x = x_0 \wedge y = y_0 \wedge x_0 \geq 0\}$
  z := 0;
  $\{$**inv** $p \equiv z = x_0 * y_0 - x*y \wedge x \geq 0\}$ $\{$**bd** x$\}$
  **while** $x \neq 0$ **do**
  **if** odd(x) **then** `z := z+y; x := x-1` **fi**; $\{p \wedge even(x)\}$
  `y := 2*y; x := x÷2`
  **od**
  $\{p \wedge x = 0\}\{z = x_0 * y_0\}$

- This is a program that implements multiplication by repeated addition and bit-shifting. (Multiplication and division by 2 correspond to left and right bit shifting respectively.) It does roughly $log_2(x_0)$ iterations.

### *Example 12: Integer Square Root*

- For another example of how a faster progress step speeds up a program, recall the integer square root problem (Example 2 earlier). The basic loop was

  {**inv** $x^2 \le n < (x+y)^2 \wedge 1 \le y$} {**bd** y}
  **while** $y \ne 1$ **do** … **od**
  {$x^2 \le n < (x+1)^2$}

- To make progress, we need to decrease y. Two obvious techniques are `y := y−1` and `y := y÷2`. Let's use `y := y÷2`, in a binary-search-like method: We test the midpoint $(x+y \div 2)^2$ against n and make it the new left or right endpoint accordingly.
- Here's a partial proof outline:

  {**inv** $0 \le x^2 \le n < (x+y)^2$} {**bd** y}
  **while** $y \ne 1$ **do**
      **if** $(x+y \div 2)^2 > n$ **then**
          {$0 \le x^2 \le n < (x+y \div 2)^2 \wedge y \div 2 < t_0$}
          y := y÷2
      **else** // $(x+y \div 2)^2 \le n$
          {$0 \le (x+y \div 2)^2 \le n < (x+y \div 2 + (y-y \div 2))^2 \wedge (y-y \div 2) < t_0$}
          x := x+y ÷ 2; y := y − y÷2
      **fi;** {$0 \le x^2 \le n < (x+y)^2 \wedge y < t_0$}
  **od**
  {$0 \le x^2 \le n < (x+y)^2 \wedge y \ge 1$} $\wedge\ y = 1$}
  {$0 \le x^2 \le n < (x+1)^2$}

- **Notes**: The invariant implies $y \ge 1$; that with $y \ne 1$ implies $y \ge 2$. That in turn implies y÷2 and y−y÷2 are both < y, which ensures progress whether the **if** test succeeds or fails.

# *Finding Invariants; Examples*

*CS 536: Science of Programming*

## *A. Why*

- It is easier to write good programs and check them for defects than to write bad programs and then debug them.
- The hardest part of programming is finding good loop invariants.
- There are heuristics for finding them but no algorithms that work in all cases.

## *B. Objectives*

At the end of this activity assignment you should

- Know how to generate possible invariants using the techniques "Replace a constant by a variable", "Drop a conjunct" or "Add a disjunct".

## *C. Questions*

1. What are the constants in the postcondition $x = max(b[0], b[1], ..., b[n-1])$? Using the technique "replace a constant by a variable," list the possible invariants for this postcondition. Also, what would the loop tests be? (Assume $n-1$ is a constant.)
2. Repeat, on the postcondition $x = n!$ (where $n!$ is short for $1*2*3*...*n$).
3. Repeat, on the postcondition $\forall i . 0 \le i < n \rightarrow b[i] = 3$.
4. Repeat, on the postcondition $\forall i . \forall j . 0 \le i < K \land K \le j < n \rightarrow b[i] < b[j]$. (Every value in $b[0...K-1]$ is < every value in $b[K...n-1]$.)
5. Consider the postcondition $x^2 \le n < (x+1)^2$, which is short for $x^2 \le n \land n < (x+1)^2$. List the possible invariant/loop test combinations you can get for this postcondition using the technique "Drop a conjunct."
6. Why is the technique "Drop a conjunct" a special case of "Add a disjunct"?
7. One way to view a search is as follows:

   {**inv** we *have* found it ∨ we *haven't* found it}
   **while** we haven't found it
   **do**
       Remove something or somethings from the things to look at
   **od**

   For this problem, try to recast (a) linear search and (b) binary search of an array using this framework: What parts of that program correspond to "we have found it", "we haven't found it", and "Remove something…"?
8. In Example 12 (integer square root), in the false branch of the **if-else** statement, can we replace the assignment $y := y - y \div 2$ with $y := y \div 2$ ? If not, why not?
9. Complete the annotation of Binary Search version 1 (Example 7).
10. Complete the annotation of Binary Search version 2 (Example 8).

 

***Solution to Activity 19 (Finding Invariants; Examples)***

1. Certainly 0 is a constant; if we replace it by a variable i, we get

   $\{$**inv** $x = max(b[i], ..., b[n-1]) \land 0 \le i \le n-1\}$ **while** $i \ne 0$ **do** …

   As a constant, n−1 seems better than just n or 1 by themselves:

   $\{$**inv** $x = max(b[0], ..., b[j]) \land 0 \le j \le n-1\}$ **while** $j \ne n\text{-}1$ **do** …

   If you want to treat just n as a constant and replace it by a variable j, we get

   $\{$**inv** $x = max(b[0], ..., b[j-1]) \land 1 \le j \le n\}$ **while** $j \ne n$ **do** …

   Similarly, if you want replace just the 1 in n−1 by with j, we get

   $\{$**inv** $x = max(b[0], ..., b[n\text{-}j]) \land 1 \le j \le n\}$ **while** $j \ne 1$ **do** …

2. We can replace n by a variable and get

   **inv** $x = i! \land 1 \le i \le n\}$ **while** $i \ne n$ **do** …

   We can replace 1 and get

   $\{$**inv** $x = j*(j+1)*\ldots*n \land 1 \le j \le n\}$ **while** $j \ne 1$ **do** …

3. For $\forall i\,.\,0 \le i < n \to b[i] = 3$ as the postcondition, we can replace 0 or n or 3.

   Replace 0 by k:

   $\{$**inv** $0 \le k \le n-1 \land \forall i\,.\,k \le i < n \to b[i] = 3\}$ **while** $k \ne 0$ **do** …

   Replace n by k

   $\{$**inv** $0 \le k \le n \land \forall i\,.\,0 \le i < k \to b[i] = 3\}$ **while** $k \ne n$ **do** …

   Replace 3 by k (this doesn't look useful)

   $\{$**inv** $\forall i\,.\,0 \le i < n \to b[i] = k\}$ **while** $k \ne 3$ **do** …

4. For $\forall i\,.\,\forall j\,.\,0 \le i < K \land K \le j < n \to b[i] < b[j]$, we have constants 0, n, and the two occurrences of K.

   Replace 0 by k:

   $\{$**inv** $0 \le k < K \land \forall i\,.\,\forall j\,.\,k \le i < K \land K \le j < n \to b[i] < b[j]\}|$

   **while** $k \ne 0$

   Replace left K by k:

   $\{$**inv** $0 \le k < K \land \forall i\,.\,\forall j\,.\,0 \le i < k \land K \le j < n \to b[i] < b[j]\}$

   **while** $k \ne K$

   Replace right K by k:

   $\{$**inv** $K \le k \le n \land \forall i\,.\,\forall j\,.\,0 \le i < K \land k \le j < n \to b[i] < b[j]\}$

   **while** $k \ne K$

   Replace n by k:

   $\{$**inv** $K \le k \le n \land \forall i\,.\,\forall j\,.\,0 \le i < K \land K \le j < k \to b[i] < b[j]\}$

   **while** $k \ne n$

   [You could argue that the ranges for k could be $0 \le k < n$, $0 \le k < n$, $0 \le k \le n$, and $0 \le k \le n$ for the four cases above; it depends on knowing more about the context of the problem.]

 

5. $\{$**inv** $n < (x+1)^2\}$ **while** $x^2 > n$ …

   $\{$**inv** $x^2 \le n\}$ **while** $n \ge (x+1)^2$ …

6. Dropping a conjunct is like adding the difference between the dropped conjunct and the rest of the predicate. For example, dropping $p_1$ from $p_1 \wedge p_2 \wedge p_3$ is like adding $(\neg p_1 \wedge p_2 \wedge p_3)$ to $(p_1 \wedge p_2 \wedge p_3)$.

7. (Rephrasing searches)

   a. We can rephrase linear search through an array with

      We have found it: $k < n \wedge b[k] = x$

      We haven't found it: $k < n \wedge b[k] \ne x$

      Remove what we're looking at from the things to look at: $k := k+1$

   b. We can rephrase binary search through an array with

      We have found it: $R = L+1$

      We haven't found it: $R > L+1$

      Remove the left or right half from the things to look at: Either $L := m$ or $R := m$

8. We can't replace $y := y - y \div 2$ by $y := y \div 2$ when $y$ is odd because then $y \div 2 = y - y \div 2 - 1$, which is not strong enough to re-establish $n < (x+y)^2$.

9. (Binary search, version 1) [Not included: The intermediate conditions within loop initialization]

   (To cut down on the writing, I'm using "f" for "found" below.)

   $\{q_0 \equiv \mathrm{Sorted}(b, n) \wedge n \ge 1 \wedge b[0] \le x < b[n]\}$

   $L := 0\ ;\ R := n\ ;\ f := F\ ;$

   $\{\mathrm{Sorted}(b, n) \wedge n \ge 1 \wedge b[0] \le x < b[n] \wedge L = 0 \wedge R = n \wedge f = F\}$

   $\{$**inv** $p \equiv 0 \le L < R \le n \wedge b[L] \le x < b[R] \wedge (f \rightarrow x = b[L])\}$ $\{$**bd** $R-L\}$

   **while** $\neg f \wedge R \ne L+1$ **do**

   $\{p \wedge \neg f \wedge R \ne L+1 \wedge R-L = t_0\}$

   $m := (L+R)/2\ ;$

   $\{p_1 \equiv p \wedge \neg f \wedge R \ne L+1 \wedge R-L = t_0 \wedge m = (L+R)/2\}$

   **if** $b[m] = x$ **then**

   $\{p_1 \wedge b[m] = x$

   $\equiv 0 \le L < R \le n \wedge b[L] \le x < b[R] \wedge (f \rightarrow x = b[L])$

   $\wedge \neg f \wedge R \ne L+1 \wedge R-L = t_0 \wedge m = (L+R)/2 \wedge b[m] = x\}$

   $\{p[T/f][m/L] \wedge R-m < t_0$

   $\equiv 0 \le m < R \le n \wedge b[m] \le x < b[R] \wedge (T \rightarrow x = b[m]) \wedge R-m < t_0\}$

   $f := T\ ;\ L := m$

   $\{p \wedge R-L < t_0\}$

   **else if** $b[m] < x$ **then**

   $\{p_1 \wedge b[m] < x$ // technically, should include $b[m] \ne x$

$\equiv 0 \le \mathtt{L} < \mathtt{R} \le \mathtt{n} \wedge \mathtt{b[L]} \le \mathtt{x} < \mathtt{b[R]} \wedge (\mathtt{f} \to \mathtt{x} < \mathtt{b[L]})$

$\wedge \neg \mathtt{f} \wedge \mathtt{R} \ne \mathtt{L+1} \wedge \mathtt{R-L} = \mathtt{t}_0 \wedge \mathtt{m} = (\mathtt{L+R})/2 \wedge \mathtt{b[m]} < \mathtt{x}\}$

$\{p[\mathtt{m}/\mathtt{L}] \wedge \mathtt{R-m} < \mathtt{t}_0$

$\equiv 0 \le \mathtt{m} < \mathtt{R} \le \mathtt{n} \wedge \mathtt{b[m]} \le \mathtt{x} < \mathtt{b[R]} \wedge (\mathtt{f} \to \mathtt{x} = \mathtt{b[m]}) \wedge \mathtt{R-m} < \mathtt{t}_0\}$

**L := m**

$\{p \wedge \mathtt{R-L} < \mathtt{t}_0\}$

**else** // $\mathtt{b[m]} > \mathtt{x}$

$\{p_1 \wedge \mathtt{b[m]} > \mathtt{x}$ // technically, should include $\mathtt{b[m]} \ne \mathtt{x} \wedge \mathtt{b[m]} \not< \mathtt{x}$

$\equiv 0 \le \mathtt{L} < \mathtt{R} \le \mathtt{n} \wedge \mathtt{b[L]} \le \mathtt{x} < \mathtt{b[R]} \wedge (\mathtt{f} \to \mathtt{x} < \mathtt{b[L]})$

$\wedge \neg \mathtt{f} \wedge \mathtt{R} \ne \mathtt{L+1} \wedge \mathtt{R-L} = \mathtt{t}_0 \wedge \mathtt{m} = (\mathtt{L+R})/2 \wedge \mathtt{b[m]} > \mathtt{x}\}$

$\{p[\mathtt{m}/\mathtt{R}] \wedge \mathtt{m-L} < \mathtt{t}_0$

$\equiv 0 \le \mathtt{L} < \mathtt{m} \le \mathtt{n} \wedge \mathtt{b[L]} \le \mathtt{x} < \mathtt{b[m]} \wedge (\mathtt{f} \to \mathtt{x} = \mathtt{b[L]}) \wedge \mathtt{m-L} < \mathtt{t}_0\}$

**R := m**

$\{p \wedge \mathtt{R-L} < \mathtt{t}_0\}$

**fi fi**

$\{p \wedge \mathtt{R-L} < \mathtt{t}_0\}$

**od**

$\{p \wedge (\mathtt{f} \vee \mathtt{R} = \mathtt{L+1})\}$

$\{0 \le \mathtt{L} < \mathtt{n} \wedge (\mathtt{f} \leftrightarrow \mathtt{x} = \mathtt{b[L]})\}$

10. (Binary search, version 2) [Not included: The intermediate conditions within loop initialization]
(To cut down on the writing, I'm using "f" for "found" below.)

$\{\mathtt{n} > 0 \wedge \mathtt{Sorted}(\mathtt{b}, \mathtt{n}) \wedge \mathtt{b[0]} \le \mathtt{x} < \mathtt{b[n-1]}\}$

**L := 0; R := n−1; f := F;**

$\{\mathtt{n} > 0 \wedge \mathtt{Sorted}(\mathtt{b}, \mathtt{n}) \wedge \mathtt{b[0]} \le \mathtt{x} < \mathtt{b[n-1]} \wedge \mathtt{L} = 0 \wedge \mathtt{R} = \mathtt{n-1} \wedge \mathtt{f} = \mathtt{F}\}$

$\{\mathbf{inv}\ q \equiv -1 \le \mathtt{L-1} \le \mathtt{R} < \mathtt{n} \wedge (\mathtt{f} \to \mathtt{b[L]} = \mathtt{x}) \wedge (\mathtt{x} \in \mathtt{b[0..n-1]} \leftrightarrow \mathtt{x} \in \mathtt{b[L..R]})\}$

$\{\mathbf{bd}\ \mathtt{R-L+1} + |\neg \mathtt{f}|\}$

**while** $\neg \mathtt{f} \wedge \mathtt{L} \le \mathtt{R}$ **do**

$\{q \wedge \neg \mathtt{f} \wedge \mathtt{L} \le \mathtt{R} \wedge \mathtt{R-L+1} + |\neg \mathtt{f}| = \mathtt{t}_0\}$

**m := (L+R)/2;**

$\{q_1 \equiv q \wedge \neg \mathtt{f} \wedge \mathtt{L} \le \mathtt{R} \wedge \mathtt{R-L+1} + |\neg \mathtt{f}| = \mathtt{t}_0 \wedge \mathtt{m} = (\mathtt{L+R})/2\}$

**if** $\mathtt{b[m]} = \mathtt{x}$ **then**

$\{q_1 \wedge \mathtt{b[m]} = \mathtt{x}$

$\equiv -1 \le \mathtt{L-1} \le \mathtt{R} < \mathtt{n} \wedge (\mathtt{f} \to \mathtt{b[L]} = \mathtt{x}) \wedge (\mathtt{x} \in \mathtt{b[0..n-1]} \leftrightarrow \mathtt{x} \in \mathtt{b[L..R]})$

$\wedge \neg \mathtt{f} \wedge \mathtt{L} \le \mathtt{R} \wedge \mathtt{R-L+1} + |\neg \mathtt{f}| = \mathtt{t}_0 \wedge \mathtt{m} = (\mathtt{L+R})/2 \wedge \mathtt{b[m]} = \mathtt{x}\}$

$\{q[\mathtt{T}/\mathtt{f}]\ [\mathtt{m}/\mathtt{L}] \wedge \mathtt{R-(m+1)+1} + |\neg \mathtt{T}| < \mathtt{t}_0$

$\equiv -1 \le \mathtt{m-1} \le \mathtt{R} < \mathtt{n} \wedge (\mathtt{T} \to \mathtt{b[m]} = \mathtt{x})$

$\wedge (\mathtt{x} \in \mathtt{b[0..n-1]} \leftrightarrow \mathtt{x} \in \mathtt{b[m..R]}) \wedge \mathtt{R-m+1} + |\neg \mathtt{T}| < \mathtt{t}_0\}$

**f := T ; L := m**

$\{q \wedge \mathtt{R-L+1} + |\neg \mathtt{f}| < \mathtt{t}_0\}$

**else if** b[m] < x **then**

$\{q_1 \wedge$ b[m] < x    // technically, should include b[m] ≠ x

$\equiv -1 \le L-1 \le R < n \wedge (f \to b[L] = x) \wedge (x \in b[0..n-1] \leftrightarrow x \in b[L..R])$

$\wedge \neg f \wedge L \le R \wedge R-L+1+|\neg f| = t_0 \wedge m = (L+R)/2 \wedge b[m] < x\}$

$\{q[m+1/L] \wedge R-(m+1)+1+|\neg f| < t_0$

$\equiv -1 \le (m+1)-1 \le R < n \wedge (f \to b[m+1] = x)$

$\wedge (x \in b[0..n-1] \leftrightarrow x \in b[m+1..R]) \wedge R-(m+1)+1+|\neg f| < t_0\}$

L := m+1

$\{q \wedge R-L+1+|\neg f| < t_0\}$

**else** // b[m] > x  // technically, should include b[m] ≠ x ∧ b[m] ≮ x

$\{q_1 \wedge b[m] > x$

$\equiv -1 \le L-1 \le R < n \wedge (f \to b[L] = x) \wedge (x \in b[0..n-1] \leftrightarrow x \in b[L..R])$

$\wedge \neg f \wedge L \le R \wedge R-L+1+|\neg f| = t_0 \wedge m = (L+R)/2 \wedge b[m] > x\}$

$\{q[m-1/R] \wedge (m-1)-L+1+|\neg f| < t_0\}$

R := m-1

$\{q \wedge R-L+1+|\neg f| < t_0\}$

**fi fi** $\{q \wedge R-L+1+|\neg f| < t_0\}$

**od**

$\{q \wedge (f \vee L > R)$

$\equiv -1 \le L-1 \le R < n \wedge (f \to b[L] = x) \wedge (x \text{ in } b[0..n-1] \leftrightarrow x \text{ in } b[L..R])$

$\wedge (f \vee L > R)\}$

$\{-1 \le L-1 \le R < n \wedge (f \to b[L] = x) \wedge (\neg f \to x \notin b[0..n-1])\}$