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 class you should

- Know how to generate possible invariants using “Drop a conjunct” or “Add a disjunct” and to be familiar with some examples of these techniques.

C. Finding Invariants - Review

- An invariant needs to be easy to establish with initialization code, it needs to establish the postcondition (when the loop test fails), and it has to be an approximation to the \( sp \) and \( wp \) of the loop body.
- There exist various general heuristics for finding invariants, though no heuristic works easily in every situation. The general idea is to weaken the postcondition somehow; the kind of weakening determines the loop test.
- **Replacing a Constant by a Variable** involves finding a constant (literal or named) \( c \) that appears in the postcondition \( q \) and replacing it by a new variable \( x \). If \( invariant[c/x] = postcondition \), then the loop test is \( \texttt{while} \ x \neq c \). The invariant has to be easily established by assigning \( x \) (and any other loop variables) to some initial value, and we have to be able to approach terminal of the loop by making \( x \) “closer” to \( c \).
- **Replacing a Constant by a Variable** is an instance of a more general technique, Adding or Modifying Parameters, in which we replace an expression by an expression that includes one or more new variables. Examples of these two included summation, factorial, integer square root, and integer log base 2.
D. Deleting A Conjoint

- 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 $p_1 \land p_2 \ldots \land p_n$ where $n \geq 2$.
- Let, $\text{Less}(r, k) = (p_1 \land p_2 \ldots \land p_{k-1}) \land (p_{k+1} \land \ldots \land p_n)$. I.e., $r$ “less” the conjunct $p_k$.
- There are $n$ possible invariants, one for each conjunct. In general, for conjunct $k$ we have
  \[
  \{ \text{inv } p = \text{Less}(r, k) \}
  \]
  \[
  \text{while } \neg p_k \text{ do}
  \]
  \[
  \{ p \land \neg p_k \} \ldots \{ p \}
  \]
  \[
  \text{od}
  \]
  \[
  \{ p \land p_k \} \{ r \}
  \]

Example 1: Linear Search of an Array

- Precondition: Array $b$ has at least $n$ elements ($n \geq 0$) 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
  \[
  0 \leq k \leq n \land x \notin b[0..k-1] \land (k < n \rightarrow b[k] = x)
  \]
  where $x \notin b[0..k-1]$ means $\forall 0 \leq k < k. x \neq b'[k]$. Note if $k = 0$, then $b[0..k-1] = b[0..-1]$ is the empty sequence of values.
- Since $0 \leq k \leq n$ is short for $0 \leq k \land k \leq n$, 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.
- Dropping the first conjunct, $0 \leq k$, forces $k < 0$ in the loop body, which makes referencing $b[k]$ illegal. This sounds really unpromising.
  \[
  \{ \text{inv } k \leq n \land x \notin b[0..k-1] \land (k < n \rightarrow b[k] = x) \} \text{ while } 0 > k \text{ do } \ldots
  \]
- Dropping $k \leq n$ has the symmetric problem: $k > n$ in the loop body makes $b[k]$ erroneous.
  \[
  \{ \text{inv } 0 \leq k \land x \notin b[0..k-1] \land (k < n \rightarrow b[k] = x) \} \text{ while } k > n \text{ do } \ldots
  \]
- Dropping the conjunct $x \notin b[0..k-1]$ causes two problems. First, how do we initialize $k$? To initialize $k := 1$ requires $b[0] = x$ whereas initializing $k := n$ 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.
  \[
  \{ \text{inv } 0 \leq k \leq n \land (k < n \rightarrow b[k] = x) \} \text{ while } x \in b[0..k-1] \text{ do } \ldots
  \]
- The fourth possibility, however, works well. Here, we drop $k < n \rightarrow b[k] = x$.
  \[
  \{ \text{inv } 0 \leq k \leq n \land x \notin b[0..k-1] \} \text{ while } -(k < n \rightarrow b[k] = x) \text{ do } \ldots
  \]
- Let’s borrow the short-circuiting && operator from C: if $e_1$ and $e_2$ are boolean expressions then $e_1 \land e_2 = \text{if } e_1 \text{ then } e_2 \text{ else } F \text{ fi}$.
- Now we can rewrite $-(k < n \rightarrow b[k] = x)$ as $k < n \land b[k] \neq x$.  

© J. Sasaki 2021
• Initialization is easy: \( k := 0 \), since its \( wp \) is \( 0 \leq k \leq n \land x \notin b[0..0–1] \). The only nontrivial part is \( n \geq 0 \), 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

\[
\begin{align*}
\{ p \land k < n \land b[k] \neq x \} & \quad // \text{Invariant} \land \text{loop test} \\
\{ 0 \leq k+1 \leq n \land x \notin b[0..k+1–1] \} & \quad // \text{wp of progress step} \\
\{ k := k+1 \} & \quad // \text{Progress step} \\
\{ 0 \leq k \leq n \land x \notin b[0..k–1] \} & \quad // \text{Invariant}
\end{align*}
\]

where ??? is code that can must us from the precondition of the loop body to the \( wp \) of the loop body. But it turns out that we don't need any code to do this.

• Convergence is easy: Since \( p \) includes \( k \leq n \) and \( k \) gets incremented, we can use \( n–k \). So the whole loop is

\[
\begin{align*}
\{ n \geq 0 \} \quad & k := 0; \\
\{ \text{inv } p = 0 \leq k \leq n \land x \notin b[0..k–1] \} \{ \text{bd } n–k \} \\
\text{while } & k < n \land b[k] \neq x \{ \text{loop body } p \land q \} \\
\text{do } & \{ 0 \leq k \leq n \land x \notin b[0..k–1] \} \land (k < n \implies b[k] = x) \}
\end{align*}
\]

E. 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

\[
\begin{align*}
\{ \text{inv } r \lor B \} \\
\text{while } B \text{ do} \\
\{ (r \lor B) \land B \} \text{ Loop body } \{ r \lor B \} \\
\text{od } \{ (r \lor B) \land \neg B \} \{ r \}
\end{align*}
\]

• 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 \( i = n \) to \( i \leq n \) (i.e., \( i = n \lor i < n \)). This is one way to understand a loop like \{inv \( i \leq n \) \} \text{ while } i < n \text{ do } \{ \text{od } i = n \}: The postcondition \( i = n \) gets the disjunct \( i < n \) added and becomes \( i \leq n \) in the invariant.

• Adding a disjunct is one way to view deleting a conjunct: Changing \( p \land q \) to \( p \lor (p \land \neg q) \)

yields something \( \Leftrightarrow \) just \( p \).

• Converting \( p \land q \) to \( p \lor q \) can be viewed as a generalization of \( \land \) to \( \lor \) or as taking \( p \land q \) to \( (p \land q) \lor (p \land \neg q) \lor q \).
F. Example 2: 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.
- Binary search has a small subtlety about what to do when the left and right endpoints are adjacent: Because of integer division, midpoint = (L+(L+1))/2 = L, which is a problem if you're claiming that the distance from L to the midpoint always decreases.
  - We'll see a couple of ways to handle this: The first one uses a sentinel value, and when R = L+1, the loop halts regardless of finding the value or not. The second one allows R = L+1, and R < L causes halting.
  - The differences between the two approaches makes the postconditions different, which in turn makes the invariants different, and (as it turns out) the loop test is different.

Binary Search version 1

- Program specification: \( \{ q_0 \} \text{Binsearch}(b, x, n) \{ r \} \) where
  - \( q_0 = \text{Sorted}(b, n) \land 1 \leq n < |b| \land b[0] \leq x < b[n] \) (writing \( |b| \) for size(b))
  - \( \text{Sorted}(b, n) = \forall 0 \leq k < n-1 < |b| \land b[k] \leq b[k+1]. \)
  - \( r = 0 \leq L < n \land b[L] \leq x < b[L+1] \land (\text{found} \leftrightarrow x = b[L]) \)
- Having \( x < b[n] \) means \( b[n] \) is a sentinel value, not an actual data value.
- Since \( b \) and \( n \) are named constants, \( \text{Sorted}(b, n) \) holds throughout the program, so we'll omit explicitly writing it in the conditions.
- For our invariant, let's generalize the loop precondition \( b[L] \leq x < b[L+1] \) to \( b[L] \leq x < b[R] \) where \( 0 \leq L < R \leq n \). (We replaced the expression \( L+1 \) by \( R \).) In addition, let's weaken the postcondition's \( (\text{found} \leftrightarrow x = b[L]) \) to just implication: \( (\text{found} \rightarrow x = b[L]) \); this lets us have \( \text{found} = F \) while we search. For the bound function, we can use \( R-L \); it's a loose termination bound but that's okay.
- For the loop body, we'll begin by calculating the midpoint \( m := (L+R)/2 \) (with truncating division). The search succeeds if \( b[m] = x \); we can set \( \text{found} \) to true and \( L \) to \( m \) and exit the loop.
• The loop so far is

\[
\{q = \text{Sorted}(b, n) \land n \geq 1 \land b[0] \leq x < b[n]\}
\]

\[
L := 0 ; R := n ; \text{found} := F ;
\]

\[
\text{inv } p = 0 \leq L < R \leq n \land b[L] \leq x < b[R] \land (\text{found} \rightarrow x = b[L])
\]

\[
\text{bd } R-L \}
\]

while \neg \text{found} \land R \neq L+1 do

\[
{p \land \neg \text{found} \land R \neq L+1 \land R-L = t_0}
\]

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

\[
{p_1 \land \neg \text{found} \land R \neq L+1 \land R-L = t_0 \land m = (L+R)/2}
\]

if \(b[m] = x\) then

\[
\{p_1 \land b[m] = x\} \text{ found} := T ; L := m ; R := L+1 \{p \land R-L < t_0\}
\]

else

\[
\{p_1 \land b[m] \neq x\} \ldots \text{ code to be filled in } \ldots \{p \land R-L < t_0\}
\]

fi

\[
\{p \land R-L < t_0\}
\]

od

\[
\{p \land (\text{found} \lor R = L+1)\}
\]

\[
\{0 \leq L < n \land b[L] \leq x < b[L+1] \land (\text{found} \leftrightarrow x = b[L])\}
\]

• It's easy to verify that loop initialization is correct. Loop termination is also correct: 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.

• The loop body calculates the midpoint \(m\) and checks \(b[m]\) against \(x\). If \(b[m] = x\), the search has succeeded and we set \(L, R,\) and found accordingly.

• If \(b[m] \neq x\), then there are two ways to make progress toward termination: \(L := m\) and \(R := m\). Both assignments have \(p \land R-L < t_0\) as postcondition, so we can calculate the wp of each assignment and see if the current precondition \(p_1 \land b[m] \neq x\) is sufficient.

• We get

\[
\{p_1 \land b[m] \neq x\} \ldots \{p[m/L] \land R-m < t_0\} \ L := m \{p \land R-L < t_0\}
\]

\[
\{p_1 \land b[m] \neq x\} \ldots \{p[m/R] \land m-L < t_0\} \ R := m \{p \land R-L < t_0\}
\]

• Expanding,

• \(p_1 \land b[m] \neq x = p \land \neg \text{found} \land R \neq L+1 \land R-L = t_0 \land m = (L+R)/2 \land b[m] \neq x\)

• \(p = 0 \leq L < R \leq n \land b[L] \leq x < b[R] \land (\text{found} \rightarrow x = b[L])\)

• \(p[m/L] = 0 \leq m < R \leq n \land b[m] \leq x < b[R] \land (\text{found} \rightarrow x = b[m])\)

• \(p[m/R] = 0 \leq L < m \leq n \land b[L] \leq x < b[m] \land (\text{found} \rightarrow x = b[L])\)

• Comparing (and omitting detailed calculations), we see that to imply the wp of \(L := m\), the precondition \(p_1 \land b[m] \neq x\) is not strong enough. We need

\[
(p_1 \land b[m] \neq x) \land (m < R \land b[m] \leq x \land R-m < t_0)
\]

• Similarly, to imply the wp of \(R := m\), we need
We can determine $b[m] < x$ and $b[m] > x$ with a test (we already know $b[m] \neq x$).

All four of $m < R$, $R-m < t_0$, $L < m$, and $m-L < t_0$ follow from $L < m < R$ and $R \neq L+1$, so there's no extra work here.

(Quick argument for $L < m < R$: Since $L+2 \leq R$, $m = (L+R)/2$ is $\geq (2*L+2)/2 = L+1$ and also $\leq (2*R-2)/2 < R$.)

Adding the test for $b[m] < x$ or $b[m] > x$ gives us a loop body partially outlined* as

```plaintext
{q ≡ Sorted(b, n) ∧ n ≥ 1 ∧ b[0] ≤ x < b[n]}
L := 0 ; R := n ; found := F ;
{inv p ≡ 0 ≤ L < R ≤ n ∧ b[L] ≤ x < b[R] ∧ (found → x = b[L])}
{bd R–L} while ¬found ∧ R ≠ L+1 do
  {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}
```

```plaintext
od
{p ∧ (found ∨ R = L+1)}
```

```plaintext
{0 ≤ L < n ∧ b[L] ≤ x < b[L+1] ∧ (found ↔ x = b[L])}
```

G. Example 3: 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, $Sorted(b, n) ∧ n ≥ 1 ∧ b[0] ≤ x ≤ b[n]$. (We weakened $x < b[n]$ to $x ≤ b[n]$.)

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,

$$-1 ≤ L–1 ≤ R < n ∧ (found → b[L] = x) ∧ (¬found → x \notin b[0..n–1])$$

(The first conjunct, $-1 ≤ L–1 ≤ R < n$, summarizes the properties and relationships of $L$ and $R$, namely $0 ≤ L < n$ and either $L ≤ R < n$ or $R = L–1$.)

* A nice at-home activity is to completely expand the annotation.
• For the invariant, we want to weaken \( \neg \text{found} \rightarrow x \not\in b[0..n-1] \) to something that will be true during the search. We only change \( L \) and \( R \) in ways that don’t alter “Is \( x \) in \( b[L..R] \)?” I’ll use \( (x \in b[L..R] \leftrightarrow x \in b[0..n-1]) \) with the understanding that when \( R = L-1 \) then \( b[L..R] = b[L..L-1] = \varnothing \). This way, if \( R < L \), we know the search has failed. We should terminate the loop if \( \text{found} \) or \( (R < L \land \neg \text{found}) \).

• 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 \( \text{find} \) \( b[m] = x \), all we do is set \( \text{found} := \text{true} \) and \( L := m \), which doesn’t necessarily decrease \( R-L+1 \). To take \( \text{found} \) into account, define \( |F| = 0 \) and \( |T| = 1 \), then the bound function can be \( R-L+1+|\neg \text{found}| \).

• Altogether, we get the following sketch for our binary search:

\[
\{ n > 0 \land \text{Sorted}(b, n) \land b[0] \leq x \leq b[n-1] \}
\]

\[
L := 0; R := n-1; \text{found} := F;
\]

\[
\{ \text{inv } q \equiv -1 \leq L-1 \leq R < n \land (\neg \text{found} \rightarrow b[L] = x) \land (x \in b[0..n-1] \leftrightarrow x \in b[L..R]) \}
\]

\[
\{ \text{bd } R-L+1+|\neg \text{found}| \}
\]

\[
\text{while } \neg \text{found} \land L \leq R \text{ do}
\]

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

\[
\{ q \equiv q \land \neg \text{found} \land L \leq R \land R-L+1+|\neg \text{found}| = t_0 \land m = (L+R)/2 \}
\]

\[
\text{if } b[m] = x \text{ then}
\]

\[
\text{found} := T; L := m
\]

\[
\text{else if } b[m] < x \text{ then}
\]

\[
L := m+1
\]

\[
R := m-1
\]

\[
\}
\]

\[
\text{fi fi}
\]

\[
\text{od}
\]

\[
\{ q \land (\neg \text{found} \lor L > R) \}
\]

\[
\{ -1 \leq L-1 \leq R < n \land (\text{found} \rightarrow b[L] = x) \land (\neg \text{found} \rightarrow x \not\in b[0..n-1]) \}
\]

**Example 4: 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 \( i = n \land j = m \).

• 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 \( 0 \leq i \leq n \land 0 \leq j \leq m \).

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

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

• 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 \( b_2[j] \):

\[
\text{NoMatch}(i, j) = \forall 0 \leq i' < i \land \forall 0 \leq j' < j \land (i' < i \land j' < j \land b_1[i'] \neq b_2[j'])
\]
• Also, let \( InRange(i, j) = 0 \leq i \leq n \land 0 \leq j \leq m \), then our postcondition is
  \[
  q = InRange(i, j) \land NoMatch(i, j) \land (i < n \land j < m \rightarrow b_1[i] = b_2[j])
  \]

• To get an invariant, we'll drop the third conjunct \((i < n \land j < m \rightarrow b_1[i] = b_2[j])\) (or, equivalently, add a disjunct of \((i < n \land j < m \rightarrow b_1[i] \neq b_2[j])\)).

  \[
  \{\text{inv } p = InRange(i, j) \land NoMatch(i, j)\}
  \]

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

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

As in linear search (Example 1), we'll rewrite the test as \( B = (i < n \land j < m \& \& b_1[i] \neq b_2[j]) \). As a conditional expression, this is \( \text{if } i < n \land j < m \text{ then } b_1[i] \neq b_2[j] \text{ else } F \ fi \).

• 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.

  \[
  \{n \geq 0 \land m \geq 0 \land Sorted(b, n) \land Sorted(b_2, m)\}
  \]

  \[
  i := 0; j := 0 \{\text{Range}(0, 0) \land NoMatch(0, 0)\}
  \]

  \[
  \{\text{inv } p = InRange(i, j) \land NoMatch(i, j)\} \{bd (n-i) + (m-j)\}
  \]

  \[
  \text{while } i < n \land j < m \& \& b_1[i] \neq b_2[j] \text{ do } \ldots \text{ od}
  \]

  \[
  \{q = p \land B\} \quad // \text{where } B = i < n \land j < m \rightarrow b_1[i] = b_2[j]\]

• For termination, we need the invariant to imply \((n-i) + (m-j) \geq 0\), 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

  \[
  \{p \land \neg B\} \quad \{InRange(i+1, j) \land NoMatch(i+1, j)\} \quad i := i + 1 \{p\}
  \]

  \[
  \{p \land \neg B\} \quad \{InRange(i, j+1) \land NoMatch(i, j+1)\} \quad j := j + 1 \{p\}
  \]

  \[
  \text{(Recall } p = InRange(i, j) \land NoMatch(i, j) \text{ and } \neg B \Rightarrow i < n \land j < m \& \& b_1[i] \neq b_2[j]\).}
  \]

  \[
  InRange(i, j) \land \ldots i < n \land j < m \ldots \text{ implies } InRange(i+1, j) \text{ and } InRange(i, j+1).
  \]

• So the question for \( i := i + 1 \) is how to get from \( NoMatch(i, j) \land b_1[i] \neq b_2[j] \) to \( NoMatch(i+1, j) \)? Some logic tells us that if we assume \( p \land \neg B \), 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 \( b_1[i] \neq b_2[j] \).

  • Altogether, we get \( \{p \land \neg B\} \text{ if } b_1[i] > b_2[j] \rightarrow \{p[i+1/i]\} \quad i := i + 1 \text{ fi} \{p\} \) as one (nondeterministic) case for the loop body.

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

    \[
    \{p \land \neg B\} \text{ if } b_2[j] > b_1[i] \rightarrow \{p[j+1/j]\} \quad j := j + 1 \text{ fi} \{p\}
    \]

• If we combining these two cases with nondeterministic \textbf{if-fi}, we get the (pleasingly?)

  • symmetric

    \[
    \{p \land \neg B\} \text{ if } b_1[i] > b_2[j] \rightarrow \{p[i+1/i]\} \quad i := i + 1
    \]
To get an invariant, we can reframe the definition of our specification and add a third array. For example:

Example 5: Multiply Integers \( x \) and \( y \) (version 1: Slowly)

- Our specification is \( \{ x = x_0 \land y = y_0 \} \ S \{ z = x_0 \cdot y_0 \} \). (\( x_0 \) and \( y_0 \) are the initial values of \( x \) and \( y \).)
  - When the loop ends, we want \( z = x_0 \cdot y_0 \).
  - When the loop begins, we have \( x_0 \cdot y_0 = x \cdot y \) because \( x = x_0 \land y = y_0 \).
- To get an invariant, we can reframe the definition of \( z \) so that it covers both cases: \( z = x_0 \cdot y_0 - x \cdot y \).
  - When the loop begins, \( x = x_0 \) and \( y = y_0 \), so \( x_0 \cdot y_0 = x \cdot y \), so we'll set \( z := 0 \).
  - We can end the loop if \( x \) or \( y = 0 \), because \( z = x_0 \cdot y_0 - x \cdot y = x_0 \cdot y_0 - 0 \).
  - If \( x_0 \geq 0 \) initially, then we can maintain \( 0 \leq x \leq x_0 \), 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 \( x \neq 0 \) as the loop test gives us

\[
\{ x = x_0 \geq 0 \land y = y_0 \} \; z := 0; \\
\{ \text{inv } p = x \geq 0 \land z = x_0 \cdot y_0 - x \cdot y \} \; \{ \text{bd } x \} \\
\text{while } x \neq 0 \\
\quad \{ p \land x \neq 0 \} \; \{ \text{code to write } \} \\
\quad \{ w \} \; x := x-1 \; \{ p \} \quad \text{ // where } w = wp(x := x-1, \; p) \\
\quad \{ p \land x = 0 \} \; z := 0; \\
\} \\
\{ \text{inv } p \equiv \{ \text{bd } x \} \} \\
\}
\]

Above, \( w = wp(x := x-1, \; p) = p[x-1/x] = (z = x_0 \cdot y_0 - (x-1) \cdot y \land x-1 \geq 0) \)

The loop body precondition \( p \land x \neq 0 = (z = x_0 \cdot y_0 - x \cdot y \land x \geq 0) \land x \neq 0 \)

Note \( p \) implies \( z = x_0 \cdot y_0 - x \cdot y \), but \( w \) requires \( z = x_0 \cdot y_0 - (x-1) \cdot y \).

- So we don't have \( p \land x \neq 0 \rightarrow w \), 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 \( (x_0 \cdot y_0 - x \cdot y) - (x_0 \cdot y_0 - (x-1) \cdot y) = x \cdot y - (x-1) \cdot y = y \)
- So \( \{ p \land x \neq 0 \} \; z := z+y \; \{ w \} \; x := x-1 \; \{ p \} \)

Our program is

\[
\{ x = x_0 \geq 0 \land y = y_0 \} \; z := 0; \\
\{ \text{inv } p = z = x_0 \cdot y_0 - x \cdot y \land x \geq 0 \} \; \{ \text{bd } x \} \\
\text{while } x \neq 0 \\
\quad \{ p \land x \neq 0 \land x = t_0 \} \; \{ p[x-1/x][z+y/z] \land x-1 < t_0 \} \\
\quad z := z+y; \; \{ p[x-1/x] \land x-1 < t_0 \} \\
\quad x := x-1 \; \{ p \land x < t_0 \} \\
\quad \{ p \land x = 0 \} \; \{ z = x_0 \cdot y_0 \} \\
\}
\]

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 \( x \geq 0 \) as a conjunct, so \( \text{invariant } \rightarrow \text{bound } \geq 0 \) holds.
- The loop body decrements \( x \), so \( \{ \text{invariant } \land \text{loop test } \land \text{bound } = t_0 \} \; \text{loop body } \{ \text{bound exp } < t_0 \} \)

holds.

### I. Example 6: 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 \( x \) as the bound expression: The invariant still implies \( x \geq 0 \), and if \( x \neq 0 \), then \( x := x/2 \) brings us strictly closer to 0.

Instead of a loop body of

\[
\{p \land x \neq 0 \land x = t_0\} \; z := z+y; \; x := x-1 \; \{p \land x < t_0\}
\]

we have

\[
\{p \land x \neq 0 \land x = t_0\} \; ?? \; \{w_1\} \; x := x+2 \; \{p \land x < t_0\}
\]

where \( w_1 = \wp(x := x+2, \; p \land x < t_0) \)

\[
= (p \land x < t_0)(x+2/x)
\]

\[
= p(x+2/x) \land x+2 < t_0
\]

\[
= (z = x_0*y_0 - (x+2)*y) \land x+2 \geq 0 \land x+2 < t_0
\]

The missing statement has to take us from \( p \land x \neq 0 \land x = t_0 \) to \( w_1 \).

We're already ensured that the \( x+2 \geq 0 \) and \( x+2 < t_0 \) clauses of \( w_1 \) hold:

\begin{itemize}
  \item \( p \) implies \( x \geq 0 \), so we know \( x+2 \geq 0 \).
  \item \( x = t_0 \) and \( x \geq 0 \land x \neq 0 \) implies \( x+2 < t_0 \).
\end{itemize}

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 \( w_1 \).

\begin{itemize}
  \item If \( x \) is even, then \( (x+2)*(2*y) = x*y \).
  \item So \( \{p \land \text{even}(x)\} \; y := 2*y; \; \{w_1\} \; x := x+2 \; \{p\} \)
\end{itemize}

But we don't know that \( x \) is even. We could check for it:

\begin{verbatim}
if even(x)
    then ... code above (requires x to be even) ... \{w_1\}
else
    \{p \land x \neq 0 \land odd(x)\} ?? \{w_1\}
fi
\end{verbatim}

Or we could force \( x \) to be even:

\begin{verbatim}
\{p\} if odd(x) then ??? ; x := x-1 fi; \{p \land even(x)\}
... above code ... \{w_1\}
\end{verbatim}

But we already know what we can use before the decrement of \( x \).

\begin{itemize}
  \item We've already written it once: it's \( z := z+y \).
\end{itemize}

This completes the program:

\begin{verbatim}
\{x = x_0 \land y = y_0 \land x_0 \geq 0\}

z := 0;

{inv p = z = x_0*y_0 - x*y \land x \geq 0} \{bd x\}

while x \neq 0 do
    if odd(x) then z := z+y; x := x-1 fi; \{p \land even(x)\}
    y := 2*y; x := x+2
od

\{p \land x = 0\} \{z = x_0*y_0\}
\end{verbatim}
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 7: Integer Square Root**

- For another example of how a faster progress step speeds up a program, recall the integer square root problem (from the previous class):

  ```
  \{ \text{inv } x^2 \leq n < (x+y)^2 \wedge 1 \leq y \} \{ \text{bd } y \}
  \text{while } y \neq 1 \text{ do } \ldots \text{ od}
  \{ x^2 \leq n < (x+1)^2 \}
  ```

- To make progress, we need to decrease $y$. Two obvious techniques are $y := y - 1$ and $y := y \div 2$. Let's use $y := y \div 2$, 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:

  ```
  \{ \text{inv } 0 \leq x^2 \leq n < (x+y)^2 \} \{ \text{bd } y \}
  \text{while } y \neq 1 \text{ do }
  \hspace{1em} \text{if } (x+y \div 2)^2 > n \text{ then }
  \hspace{2em} \{ 0 \leq x^2 \leq n < (x+y \div 2)^2 \wedge y \div 2 < t_0 \}
  \hspace{2em} y := y \div 2
  \hspace{1em} \text{else } \text{ // } (x+y \div 2)^2 \leq n
  \hspace{2em} \{ 0 \leq (x+y \div 2)^2 \leq n < (x+y \div 2 + (y - y \div 2))^2 \wedge (y - y \div 2) < t_0 \}
  \hspace{2em} x := x+y \div 2; \ y := y - y \div 2
  \hspace{1em} \text{fi; } \{ 0 \leq x^2 \leq n < (x+y)^2 \wedge y < t_0 \}
  \text{od}
  \{ 0 \leq x^2 \leq n < (x+y)^2 \wedge y \geq 1 \} \wedge y = 1 \}
  \{ 0 \leq x^2 \leq n < (x+1)^2 \}
  ```

- **Notes**: The invariant implies $y \geq 1$; that with the loop test $y \neq 1$ implies $y \geq 2$. That in turn implies $y \div 2$ and $y - y \div 2$ are both $< y$, which ensures progress whether the if test succeeds or fails.
Finding Invariants

Part 2: Deleting Conjuncts; Adding Disjuncts

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 “Drop a conjunct” and “Add a disjunct”.

C. Problems

1. Consider the postcondition \( x^2 \leq n < (x+1)^2 \), which is short for \( x^2 \leq n \land n < (x+1)^2 \). List the possible invariant/loop test combinations you can get for this postcondition using the technique “Drop a conjunct.”

2. Why is the technique “Drop a conjunct” a special case of “Add a disjunct”?

3. One way to view a search is as follows:

\[
\{ \text{inv found it } \lor \text{ not found it} \} \\
\text{while not found it} \\
\text{do} \\
\text{Remove something or somethings from the things to look at} \\
\text{od}
\]

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…”?

4. In Example 7 (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?

5. Complete the annotation of Binary Search version 1 (Example 2).

6. Complete the annotation of Binary Search version 2 (Example 3).
Solution to Activity 20 (Finding Invariants; Examples)

1. \(\{\text{inv } n < (x+1)^2\} \text{ while } x^2 > n \ldots\)
   \(\{\text{inv } x^2 \leq n\} \text{ while } n \geq (x+1)^2 \ldots\)

2. Dropping a conjunct is like adding the difference between the dropped conjunct and the rest of the predicate. E.g., dropping \(p_1\) from \(p_1 \land p_2 \land p_3\) is like adding \((\neg p_1 \land p_2 \land p_3)\) to \((p_1 \land p_2 \land p_3)\).

3. (Rephrasing searches)
   a. We can rephrase linear search through an array with
      We have found it: \(k < n \land b[k] = x\)
      We haven’t found it: \(k < n \land b[k] \neq x\)
      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 \(R := m\)

4. We can’t replace \(y := y - y/2\) by \(y := y/2\) because for \(y\) odd, \(y/2 = y - y/2 - 1\), which is not strong enough to re-establish \(n < (x+y)^2\).

5. (Binary search, version 1) [Not included: The intermediate conditions within loop initialization]
   \(\{q_0 \equiv \text{Sorted}(b, n) \land n \geq 1 \land b[0] \leq x < b[n]\}\)
   \(L := 0 ; R := n ; \text{found} := F ;\)
   \(\{\text{Sorted}(b, n) \land n \geq 1 \land b[0] \leq x < b[n] \land L = 0 \land R = n \land \neg\text{found}\}\)
   \(\{\text{inv } p = 0 \leq L < R \leq n \land b[L] \leq x < b[R] \land (\text{found} \rightarrow x = b[L])\} \{\text{bd } R-L\}\)
   \(\text{while } \neg\text{found} \land R \neq L+1 \text{ do}\)
   \(\{p \land \neg\text{found} \land R \neq L+1 \land R-L = t_0\}\)
   \(m := (L+R)/2 ;\)
   \(\{p_1 = p \land \neg\text{found} \land R \neq L+1 \land R-L = t_0 \land m = (L+R)/2\}\)
   \(\text{if } b[m] = x \text{ then}\)
   \(\{p_1 \land b[m] = x = 0 \leq L < R \leq n \land b[L] \leq x < b[R] \land (\text{found} \rightarrow x = b[L])\}\)
   \(\land \neg\text{found} \land R \neq L+1 \land R-L = t_0 \land m = (L+R)/2 \land b[m] = x\}\)
   \(\{p[T/\text{found}][m/L] \land R-m < t_0\}\)
   \(= 0 \leq m < R \leq n \land b[m] \leq x < b[R] \land (T \rightarrow x = b[m]) \land R-m < t_0\}\)
   \(\text{found} := T ; L := m ;\)
   \(\{p \land R-L < t_0\}\)
   \(\text{else if } b[m] < x \text{ then}\)
6. (Binary search, version 2) [Not included: The intermediate conditions within loop initialization]

\[ \{n > 0 \land \text{Sorted}(b, n) \land b[0] \leq x < b[n-1]\} \]

\[ L := 0; \ R := n-1; \ \text{found} := F; \]

\[ \{n > 0 \land \text{Sorted}(b, n) \land b[0] \leq x < b[n-1] \land L = 0 \land R = n-1 \land \neg \text{found}\} \]

\[ \{\text{inv} \ q = -1 \land L = 0 \land R = n \land \neg \text{found} \land x \in b[0..n-1] \land x \in b[L..R]\} \]

\[ \{\text{bd} \ R-L+1+|\neg T| \} \]

\[ \text{while} \ \neg \text{found} \land L \leq R \ do \]

\[ \{q \land \neg \text{found} \land L \leq R \land R-L+1+|\neg T| = t_0\} \]

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

\[ \{q_2 = q \land \neg \text{found} \land L \leq R \land R-L+1+|\neg T| = t_0 \land m = (L+R)/2\} \]

\[ \text{if} \ b[m] = x \ then \]

\[ \{q_3 \land b[m] = x \]

\[ = -1 \leq L-1 \leq R < n \land \neg \text{found} \land x \in b[0..n-1] \land x \in b[L..R]\} \]

\[ \land \neg \text{found} \land L \leq R \land R-L+1+|\neg T| = t_0 \land m = (L+R)/2 \land b[m] = x\} \]

\[ \{q[T] / \text{found} \ [m/L] \land R-(m+1)+1+|\neg T| < t_0\]}

\[ = -1 \leq m-1 \leq R < n \land (T \to b[m] = x) \]

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

\[ \text{found} := T; \ L := m \]

\[ \{q \land R-L+1+|\neg T| < t_0\} \]
else if $b[m] < x$ then
    \{ q, \land b[m] < x \ // technically, should include $b[m] \neq x$
    = -1 \leq L-1 \leq R < n \land (\text{found} \rightarrow b[L] = x) \land (x \in b[0..n-1] \leftrightarrow x \in b[L..R])
    \land \neg\text{found} \land L \leq R \land R-L+1+1-\neg\text{found} = t_0 \land m = (L+R)/2 \land b[m] < x\}
    \{ q[m+1/L] \land R-(m+1)+1+1-\neg\text{found} < t_0 \}
    = -1 \leq (m+1)-1 \leq R < n \land (\text{found} \rightarrow b[m+1] = x)
    \land (x \in b[0..n-1] \leftrightarrow x \in b[m+1..R]) \land R-(m+1) +1+1-\neg\text{found} < t_0\}

    L := m+1
    \{ q \land R-L+1+1-\neg\text{found} < t_0 \}
else // $b[m] > x$ // technically, should include $b[m] \neq x \land b[m] < x$
    \{ q, \land b[m] > x
    = -1 \leq L-1 \leq R < n \land (\text{found} \rightarrow b[L] = x) \land (x \in b[0..n-1] \leftrightarrow x \in b[L..R])
    \land \neg\text{found} \land L \leq R \land R-L+1+1-\neg\text{found} = t_0 \land m = (L+R)/2 \land b[m] > x\}
    \{ q(m-1/R) \land (m-1)-L+1+1-\neg\text{found} < t_0 \}
    R := m-1
    \{ q \land R-L+1+1-\neg\text{found} < t_0 \}
fi fi \{ q \land R-L+1+1-\neg\text{found} < t_0 \}

od
    \{ q \land (\text{found} \lor L > R) \}
    = -1 \leq L-1 \leq R < n \land (\text{found} \rightarrow b[L] = x) \land (x \in b[0..n-1] \leftrightarrow x \in b[L..R])
    \land (\text{found} \lor L > R) \}
    \{ -1 \leq L-1 \leq R < n \land (\text{found} \rightarrow b[L] = x) \land (\neg\text{found} \rightarrow x \notin b[0..n-1]) \}