Program Verification
Program Verification

*Program verification* is a discipline within computer science in which we prove the correctness of programs using the rules of inference and proof techniques we’ve learned so far in this class. An important goal of this work is to automate this process so that it can be done using a computer.

- A great deal of progress has been made, but we are still far away from being able to achieve the goal.
- Some mathematicians and theoretical computer scientists believe it will never be done for complex programs.
Program Verification

Definition: A program is said to be correct if it produces the correct output for every possible input.

A proof that a program is correct consists of two parts:

• Show that the correct answer is obtained if the program terminates.
• Show that the program always terminates.

To specify what it means for a program to produce the correct output, two propositions are used:

• The initial assertion gives the properties that the input values must have.
• The final assertion gives the properties that the output of the program should have if the program did what was intended.
Program Verification

Definition: A program segment S is said to be partially correct with respect to the initial assertion p and the final assertion q if, whenever p is true for the input values of S and S terminates, then q is true for the output values of S. If S is a complete program, then it is said to be correct.

We use the notation $p \{ S \} q$ to indicate that the program segment S is partially correct with respect to the initial assertion p and the final assertion q. This notation is called a Hoare triple.
Rules of Inference

Composition rule:

\[ p \{ S_1 \} q \]
\[ q \{ S_2 \} r \]
\[ \therefore p \{ S_1 ; S_2 \} r \]
Rules of Inference

**If-then rule:**

\[(p \land \text{condition}) \{ S \} q \]
\[(p \land \neg \text{condition}) \rightarrow q \]
\[\therefore p \{ \text{if condition then } S \} q\]

**If-then-else rule:**

\[(p \land \text{condition}) \{ S_1 \} q \]
\[(p \land \neg \text{condition}) \{ S_2 \} q \]
\[\therefore p \{ \text{if condition then } S_1 \text{ else } S_2 \} q\]
Rules of Inference

Definition: p is a *loop invariant* if

\[(p \land \text{condition}) \{ S \} p\]

is true.

**While rule:**

\[\begin{align*}
(p \land \text{condition}) \{ S \} p \\
\hline
\therefore \ p \{ \text{while condition} \ S \} (\neg \text{condition} \land p)
\end{align*}\]
Let $A[1..n]$ be an input array. We want to sort the elements of $A$ into increasing order.

```
for j := 2 to n {
    # incorporate the $j^{th}$ element
    x := A[j]
    i := j-1
    while (i > 0) and (A[i] > x) {
        A[i+1] := A[i]
        i := i-1
    }
    A[i+1] := x
}
```
A bag (or multiset) is a collection in which order does not matter, and duplicate items are permitted.

A majority item of a bag is one that occurs more often in a bag than all other items combined.

Given a bag B, an algorithm is needed to either find the majority item of B, or indicate that no majority item exists. The Boyer-Moore Majority Finding Algorithm solves this problem by making two passes through the bag:

• The first pass suggests a candidate for the majority item, or that no such candidate exists.

• The second pass determines whether or not the candidate is indeed a majority item.
Boyer-Moore Majority Finding Algorithm

The algorithm for the second pass is uninteresting. We will focus on the first half.

\[ b := B \]
\[ c := \{\} \quad // \text{empty bag} \]

while \( b \neq \{\} \)

pick any \( x \) from \( b \)

if \( \exists y \) in \( c \) such that \( x \neq y \)

\[ b := b - \{x\} \]
\[ c := c - \{y\} \]
else

// \( \forall y \) in \( c \), \( x = y \)
\[ b = b - \{x\} \]
\[ c = c \cup x \]
Proposition 1: If the bag has a majority item, removing two distinct items from the bag does not alter the majority item.

Proof:
Let the bag have a majority item m, and let the number of occurrences of m in B be n.

Let the number of occurrences of all other items in the bag be n'.

Then n > n' (from the definition of "majority item").

Two cases:
1. Neither of the two distinct items removed are equal to m. Then still n > n' because n' gets smaller by two.
2. One of the two distinct items is equal to m. Then still n > n' because both n and n' get smaller by one.
Proposition 2: The statement "All elements of c are identical" is an invariant.

Proof:

Initialization: c is initially empty, so this statement is trivially true.

Maintenance: in each step, either an element is removed from c, or x (which equals every item of c) is added to c. Therefore, every step preserves the invariant.

Termination: the statement is true at the end of the loop since none of the steps ever change its truth value.
Proposition 3: Given B has a majority item m, the statement
"the majority item of \( b \cup c = m \)"
is an invariant.

Proof:
Initialization: at the start, \( b \cup c = B \), and the majority item of B is m. Therefore, the majority item of \( b \cup c \) is m.

Maintenance:
• the step corresponding to the "else" clause does not alter \( b \cup c \).
• for the step corresponding to the "then" clause, we are removing two distinct items, x from b and y from c. In other words, we are removing two distinct items from \( b \cup c \). By Proposition 1, this does not alter the majority item.

Termination: the statement is true at the end of the loop since none of the steps ever change its truth value.
Proposition 4: The algorithm terminates.

Proof:

Each iteration of the loop reduces the size of b by 1; therefore, the number of iterations is exactly |B|. 
Proposition 5: Suppose $B$ has a majority item $m$. Then, at termination, $c$ consists of one or more copies of $m$.

Proof:

- From Proposition 3, the majority item of $b \cup c$ is invariant.
- At termination, $b = \emptyset$.
- Conclude that $m$ is the majority item of $c$. So if $B$ has a majority item $m$, $c$ must be nonempty, and it must have majority item $m$.
- From Proposition 2, $c$ only contains copies of $m$. 