1. You have a jar containing \( k \) red marbles and \( m \) blue marbles, where \( k + m > 0 \). You also have an unlimited supply of blue marbles off to the side. Repeat the following procedure until exactly one marble remains in the jar:

Choose (any) two marbles from the jar;
if both are blue {
    put one back into the jar;
    discard the other one;
}
else if both are red {
    put one back into the jar;
    discard the other one;
    put a (new) blue marble into the jar;
}
else { // one marble of each color was chosen
    discard the red one;
    put the blue one back into the jar;
}

(a) Give a convincing argument that the procedure necessarily terminates.

**Hint:** It is not the case that the number of marbles in the jar necessarily decreases on each iteration. However, if you assign one weight to each red marble and another weight to each blue marble, you can show that the total weight of the marbles in the jar decreases on each iteration. It is up to you to come up with appropriate weights. **End of hint.**

(b) Make the strongest statement you can regarding the color of the last marble remaining in the jar. Is it always the same, regardless of the values of \( k \) and \( m \)? Or, failing that, can it be determined from \( k \) and \( m \)? In either case, give a convincing argument, making use of the loop invariant concept.

If, on the other hand, the color of the last marble cannot be determined from \( k \) and \( m \), give an example demonstrating this. (That is, for some particular values of \( k \) and \( m \), show that one execution of the procedure yields a blue marble and another execution yields a red marble.)

**Note:** For a slight deduction in points, you may assume not only that \( k + m > 0 \) but, more strongly, that both \( k > 0 \) and \( m > 0 \). If you make this assumption, be sure to mention it.

**Solution:** As for (a), let \( n_r \) and \( n_b \) be the number of red and blue marbles in the jar, respectively, at the beginning of an arbitrary iteration of the loop. Let \( n_r' \) and \( n_b' \) be the number of red and blue marbles in the jar, respectively, at the end of that iteration. Depending upon the pair of marbles chosen, there are three possibilities:
(i) Two blue marbles are chosen. Then \( n'_r = n_r \) and \( n'_b = n_b - 1 \).

(ii) Two red marbles are chosen. Then \( n'_r = n_r - 1 \) and \( n'_b = n_b + 1 \).

(iii) One marble of each color is chosen. Then \( n'_r = n_r - 1 \) and \( n'_b = n_b \).

Note that, in all three cases, \( n'_r + n'_b \geq n_r + n_b - 1 \), which is to say that, during any iteration, the number of marbles in the jar decreases by at most one. Keeping in mind that the loop terminates when a single marble remains in the jar, we conclude that it is not possible to arrive at a situation in which the jar is empty (because that would require that the number of marbles go from two, or some higher number, down to zero on a single iteration).

Now, let \( w_r \) and \( w_b \) be the weights of red and blue marbles, respectively, measured in some unit (e.g., ounce, gram) of weight. Then the weight of the marbles in the jar at the beginning of the iteration is

\[
w = n_r \cdot w_r + n_b \cdot w_b
\]

and the weight of the marbles in the jar at the end of the iteration is

\[
w' = n'_r \cdot w_r + n'_b \cdot w_b
\]

We wish to find values for \( w_r \) and \( w_b \) that guarantee \( w - w' \geq 1 \). For cases (i) and (iii), this is trivial, as in former we get

\[
\begin{align*}
1 & \leq w - w' \\
1 & \leq (n_r \cdot w_r + n_b \cdot w_b) - (n'_r \cdot w_r + n'_b \cdot w_b) & \text{(defn of } w, w') \\
1 & \leq (n_r - n'_r) \cdot w_r + (n_b - n'_b) \cdot w_b & \text{(algebra)} \\
1 & \leq (n_r - (n_r - 1)) \cdot w_r + (n_b - (n_b - 1)) \cdot w_b & \text{(algebra)} \\
1 & \leq w_b & \text{(algebra)}
\end{align*}
\]

and in the latter, by similar reasoning, we get \( 1 \leq w_r \). That is, to ensure that the jar of marbles decreases by at least one unit of weight in cases (i) and (iii), it suffices to choose \( w_r \) and \( w_b \) to be any values greater than or equal to one.

In case (ii) we get

\[
\begin{align*}
1 & \leq w - w' \\
1 & \leq (n_r \cdot w_r + n_b \cdot w_b) - (n'_r \cdot w_r + n'_b \cdot w_b) & \text{(defn of } w, w') \\
1 & \leq (n_r - n'_r) \cdot w_r + (n_b - n'_b) \cdot w_b & \text{(algebra)} \\
1 & \leq (n_r - (n_r - 1)) \cdot w_r + (n_b - (n_b + 1)) \cdot w_b & \text{(algebra)} \\
1 & \leq w_r - w_b & \text{(algebra)} \\
w_b + 1 & \leq w_r
\end{align*}
\]

This is satisfied by choosing \( w_r = 2 \) and \( w_b = 1 \), for example. As these choices are consistent with \( w_r \geq 1 \) and \( w_b \geq 1 \) (as cases (iii) and (i) require, respectively), we have found weights for red and blue marbles that guarantee that the weight of the marbles in the jar decreases by at least one on each iteration.

Recalling from above that the jar can never become empty, we reason that the weight of the marbles in the jar can never be less than the weight of a single blue marble (which weighs less
than a red one, after all). Hence, the number of iterations must be less than the (number of units of) weight of the marbles initially in the jar.

As for (b), the last marble in the jar will be blue, except in the case that $k = 1$ and $m = 0$. To demonstrate this, first consider the case $m > 0$:

**When** $m > 0$: In this case, we claim that *The jar contains at least one blue marble* is an invariant of the loop.

Our reasoning is as follows: From the assumption $m > 0$, it follows that, initially (i.e., before the first loop iteration), there is at least one blue marble in the jar. Now suppose that, at the beginning of an arbitrary iteration, there is at least one blue marble in the jar. During that iteration, one of three things will happen (with respect to blue marbles in the jar):

(i) Two blue marbles will be removed from the jar, but one of them will be put back into the jar.

(ii) No blue marbles will be removed from the jar, but a new blue marble will be put into the jar.

(iii) One blue marble will be removed from the jar, but it will be put back into the jar.

It is clear that, in all three cases, the iteration must end with at least one blue marble in the jar.

Summarizing, we have shown that

there is at least one blue marble in the jar just prior to the first iteration, and

if there is at least one blue marble in the jar at the beginning of an iteration, there must be at least one blue marble in the jar at the end of that iteration.

This constitutes a proof, by mathematical induction (on the number of loop iterations), that the statement *The jar contains at least one blue marble* is an invariant of the loop (which is to say that it holds immediately before and after each loop iteration).

When the loop terminates, there will be one marble in the jar; the loop invariant tells us that that marble must be blue.

**When** $m = 0$ and $k = 1$: In this case the loop terminates after zero iterations with one red marble in the jar.

**When** $m = 0$ and $k > 1$: In this case, on the first iteration of the loop two red marbles will be chosen from the jar, one will be put back, and a blue marble will be inserted. From this point onward, *The jar contains at least one blue marble* will be an invariant of the loop, by the reasoning given earlier, and so the last remaining marble will be blue. (Indeed, it is as though we began with $k - 1$ red marbles and one blue one.)
2. In class, we developed a program for solving the two-color version of the Dutch National Flag problem. The program was developed using a proposed loop invariant (derived from the post-condition) as a guide.

Here you are asked to develop an alternative solution to the same problem. As in class, your program is not to modify the array except by swapping array elements. (Assume that there is a method swap such that the effect of making the call swap(a,k,j) is to swap the values in a[k] and a[j].)

The pre-condition is that every element in the array a[] satisfies exactly one among the two predicates isRed() and isBlue(). The postcondition is as indicated in this picture:

```
<table>
<thead>
<tr>
<th></th>
<th>all RED</th>
<th>all BLUE</th>
<th>&amp; &amp; 0&lt;=k&lt;=N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>k</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
```

(We use N as an abbreviation for the length of a, which in Java is written a.length.) In words, this says that 0 ≤ k ≤ N and that every element in the array segment a[0..k - 1] is Red and every element in a[k..N - 1] is Blue.

More formally, we could express this in the language of predicate logic as follows:

```
(\forall i \mid 0 \leq i < k : isRed(a[i]) \land (\forall i \mid k \leq i < N : isBlue(a[i])) \land 0 \leq k \leq N
```

The loop invariant of your program should be as suggested by this picture:

```
<table>
<thead>
<tr>
<th></th>
<th>all RED</th>
<th>?</th>
<th>all BLUE</th>
<th>&amp; &amp; 0&lt;=k&lt;=m&lt;=N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>k</td>
<td>m</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
```

In words, this says that 0 ≤ k ≤ m ≤ N and that every element in a[0..k - 1] is Red and every element in a[m..N - 1] is Blue. More formally, we can express this by

```
(\forall i \mid 0 \leq i < k : isRed(a[i]) \land (\forall i \mid m \leq i < N : isBlue(a[i])) \land 0 \leq k \leq m \leq N
```

Arrive at your solution by correctly replacing each question mark in the incomplete program below with either an expression or a sequence of statements, whichever is appropriate. The initialization of k and m should establish the invariant by making the ?-region cover the entire array. Each iteration of the loop should decrease by at least one the length of the ?-region.

**Solution:** Because the invariant requires a[0..k - 1] to contain only red values, the only way to ensure it, before any elements of a are examined, is to make 0..k - 1 an empty range. We do that by setting k to zero. Analogously, because the invariant requires a[m..N - 1] to contain only blue values, we make m..N - 1 an empty range by setting m to N (i.e., a.length).
\[ k = ?; \quad m = ?; \]

while ( ? ) {
    if (isRed(a[k]))
        { ? }
    else /* isBlue(a[k]) */
        { ? }
}

Figure 1: Incomplete Program

Now consider the loop guard. It should be clear (by viewing the pictures) that when the range \( k..m - 1 \) is empty (corresponding to the condition \( k = m \)), the loop invariant guarantees the postcondition. (Or, to put it another way, you might say that the postcondition is, essentially, a special case of the loop invariant that occurs when \( k = m \).) Thus, the loop should terminate when \( k = m \) is satisfied, which means that the appropriate loop guard is its negation, \( k \neq m \).

The goal of each iteration of a loop is to preserve the truth of its invariant while at the same time making progress towards termination. A rather obvious way of achieving the latter in this program is to decrease \( m - k \) (i.e., the length of the array segment marked by "?" in the picture of the invariant), whose initial value is \( N \) and whose value upon termination is zero (according to the initialization code and loop guard that we’ve adopted). This suggests that, during each loop iteration, either \( k \) be increased or \( m \) decreased, or both.

Keeping this in mind, we now consider how to complete the body of the loop, which, as given, determines the color of \( a[k] \) and then takes unspecified actions according to whether it is red or blue.

In the case that \( a[k] \) is red, the loop invariant tells us that \( a[0..k] \) contains only red values, which means that, by incrementing \( k \), we are left with \( a[0..k-1] \) containing only red values, as required by the invariant. Hence, we increment \( k \).

In the case that \( a[k] \) is blue, by swapping the elements at locations \( k \) and \( m - 1 \) we get that \( a[m - 1] \) is blue. From the invariant it follows that \( a[m - 1..N - 1] \) contains only blue values. Hence, by decrementing \( m \), we are left with \( a[m..N-1] \) containing only blue values, as required by the invariant. Hence, we swap and decrement.

That the loop terminates follows from the fact that the value \( m - k \) decreases by one on each iteration, and hence eventually reaches zero, at which time the loop terminates. (Indeed, because initially \( m - k = N \), we calculate that the number of loop iterations is exactly \( N \).)

The completed program is below.

One of the more common errors made by students in answering this question was to initialize \( m \) to \( N - 1 \). This is incorrect in that it falsifies the loop invariant in the case that \( a[N - 1] \) is red. (Recall that the invariant requires all elements in the segment \( a[m..N - 1] \) to be blue.)
k = 0;  m = N;   // recall that N is shorthand for a.length

while ( k != m ) {
    if ( isRed(a[k]) )
        { k = k+1; }
    else { /* isBlue(a[k]) */
            swap(a,k,m-1);
            m = m-1;
    }
}

Figure 2: Completed Program

The same students who made this mistake also chose to swap the elements at locations k and m, rather than k and m – 1, when a[k] was found to be blue. This suggests that these students were subconsciously basing their code upon a loop invariant differing slightly from the one given. Here is an illustration:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>k</th>
<th>m</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>+-----------------+-----------------+-----------------</td>
</tr>
<tr>
<td></td>
<td>all RED</td>
<td>?</td>
<td>all BLUE</td>
<td>&amp; &amp; 0 &lt;= k &lt;= m &lt;= N</td>
</tr>
</tbody>
</table>

Specifically, their (subliminal) invariant says that all elements in a[m+1..N-1] (as opposed to a[m..N-1], as in the original) must be blue. With this invariant, their choices for initializing m and for swapping array elements are correct! But using this invariant also necessitates a change in the loop guard, because using k ≠ m results in the loop terminating with the variables in the following state:

<table>
<thead>
<tr>
<th></th>
<th>m</th>
<th>k</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>+-----------------+-----------------+-----------------</td>
</tr>
<tr>
<td></td>
<td>all RED</td>
<td>?</td>
<td>all BLUE</td>
</tr>
</tbody>
</table>

You might argue that this is a perfectly acceptable state of affairs because, as the post-condition requires, all blue elements occur to the right of all red elements. (Note that this is true regardless of whether a[k] is red or blue.) Ahh, but the post-condition requires one more thing, which is that the blue segment begins at location k. Thus, if a[k] happened to be red, the postcondition would not be satisfied!

We could fix this by inserting an if statement following the loop that increments k if a[k] is red. But this would be an ugly patch! Better would be to change the loop guard to k ≠ m + 1. (To allow the values of k and m to “cross”, we would also have to tweak the invariant, changing 0 ≤ k ≤ m ≤ N to 0 ≤ k ≤ m + 1 ≤ N.)