This note proves the correctness of the following `rotate` function. The `rotate` function should shift all the elements to the right by one position, except for the last one, which should move to the front. The following algorithm accomplishes this by starting at the end of the array moving backward, swapping two neighboring elements in each iteration. The effect is that the last element of the array migrates to the front of the array.

```java
static void rotate(int[] A) {
    if (2 <= A.length) {
        for (int i = A.length - 1; i != 0; --i) {
            swap(A, i, i - 1);
        }
    } else {
    }
}
```

The proof consists of a new version of the function that includes additional `assert` statements together with extra prose that proves certain necessary conditions. The form of those `assert` statements and the necessary conditions will follow the rules of Floyd-Hoare logic (aka. Hoare logic) adapted to Java.

## 1 Defining “correct”

The first step of proving the correctness of any program or function is to obtain, come up with, or negotiate a precise definition of what it means for it to be correct. This may involve talking to the persons who will use the function, such as a client. Many times the function is an internal one that will be used by yourself, in which case you just need to decide what the function is suppose to do. For `rotate`, the idea is to shift the elements of the array by one position to the right, with the last element moving to the front. We can write this down precisely in the form of predicate, that is, a Java function that returns a `boolean`. We name it `is_rotated` and give its definition in Figure 1.

For example, in the following code the assertion `is_rotated(B_orig, B)` should always be true after the call to `rotate`.
static boolean is_rotated(int[] A_orig, int[] A) {
    if (A.length == 0) {
        return true;
    } else {
        boolean result = A[0] == A_orig[A.length - 1];
        for (int i = 0; i != A.length - 1; ++i) {
            result &= A[i + 1] == A_orig[i];
        }
        return result;
    }
}

Figure 1: Definition of correctness for rotate.

int[] B = { 1, 2, 3, 4 }
int[] B_orig = copyOf(B, 4);
rotate(B);
assert is_rotated(B_orig, B);

We import the function copyOf (any many others) from the Java standards Arrays class:
import static java.util.Arrays.*;

To prove that rotate is correct, we apply the rules of Hoare logic to the body of rotate, creating a new function we shall call rotate_proof. We can begin by placing the correctness condition at the end of the function, as an assert statement. However, because we are going to rotate the array in-place, we need to capture the original version of the array in another array that we name A_orig.

We can assume that the copyOf function does its job so that A_orig is equal to A.

1 static void rotate_proof(int[] A) {
    int[] A_orig = copyOf(A, A.length);
    assert Arrays.equals(A_orig, A);
    ...
    assert is_rotated(A_orig, A);
}

2 Reasoning about if statements

The body of rotate is an if statement. The Floyd-Hoare logic rule for an if says that whatever is true before the if (let’s call it P) is also true at the beginning of each branch. Furthermore, the condition C is true at the start of the “then” branch and C is false at the start of the “else” branch. If something (call it Q) is true by the end of both branches, then Q is true after the if statement. This rule makes the assumption that the condition C does not contain side effects,

1Unfortunately, we must qualify the equals method as in Arrays.equals even though we imported it because it is overshadowed by the equals method in the Object class. Java!
assert \( P \);
if (C) {
    assert \( P \) \&\& \( C \);
    ...
    assert \( Q \);
} else {
    assert \( P \) \&\& !\( C \);
    ...
    assert \( Q \);
} assert \( Q \);

Figure 2: The Floyd-Hoare logic rule for if statements.

such as changing a variable that is mentioned by \( P \). Figure 2 shows the template for an if statement that correctly follows the Floyd-Hoare rule.

Getting back to the rotate function, we can instantiate the if-rule with what we already know:

- we know that \( A\_\text{orig} \) is equal to \( A \) before the loop (\( P \)),
- the condition of the if statement is \( 2 \leq A\text{.length} \) (\( C \)), and
- \( \text{is\_rotated}(A\_\text{orig}, A) \) should be true after the loop (\( Q \)).

So we can fill in the assert statements as follows.

```java
static void rotate_proof(int[] A) {
    int[] A_orig = copyOf(A, A.length);
    assert Arrays.equals(copyOfRange(A_orig, 0, A.length),
                         copyOfRange(A, 0, A.length));
    if (2 <= A.length) {
        assert Arrays.equals(copyOfRange(A_orig, 0, A.length),
                             copyOfRange(A, 0, A.length))
                            \&\& 2 <= A.length;
        ...
        assert is_rotated(A_orig, A);
    } else {
        assert Arrays.equals(copyOfRange(A_orig, 0, A.length),
                             copyOfRange(A, 0, A.length))
                            \&\& !(2 <= A.length);
        assert is_rotated(A_orig, A);
    }
    assert is_rotated(A_orig, A);
}
```

Let us start with the easy branch, the else branch. It consists of two adjacent assert statements. The Floyd-Hoare logic rule for this situation is that one must prove that the first assert logically implies the second assert (Figure 3).

Recall that to prove \( P \) implies \( Q \), we assume \( P \) is true and then prove that \( Q \) is true. So in this case we assume that \( A\_\text{orig} \) and \( A \) are equal and that their length is less than 2. We need to prove that \( \text{is\_rotated}(A\_\text{orig}, A) \). Let's break this into two cases: their length is 0 or 1.
assert $P$;
assert $Q$; // Condition: $P$ implies $Q$

Figure 3: The Floyd-Hoare logic rule for adjacent assert statements

- Length 0. Then $\text{is\_rotated}(A\_orig, A)$ is true (the “then” branch of $\text{is\_rotated}$ is taken).

- Length 1. Looking at the else branch of $\text{is\_rotated}(A\_orig, A)$, we need to show that $A[0] == A\_orig[0]$, which is true because of the assumption that $A\_orig$ and $A$ are equal. The for loop inside $\text{is\_rotated}(A\_orig, A)$ does not execute because it’s condition is immediately false because $i$ is equal to $A$.length - 1.

We have completed the proof of correctness for the else branch and new move on the “then” branch.

3 Reasoning about for loops

The most difficult part of reasoning about loops is coming up with the loop invariant, which is a partial statement of correctness that remains true but “grows” from one iteration of the loop to the next. Coming up with the loop invariant often requires some careful thinking and trial-and-error. Walking through the execution of the loop with pen-and-paper on a small example is often the best tool. Consider rotate applied to the array [1,2,3,4].

| $A\_orig$ | [1,2,3,4] |
| $A$ | [1,2,3,4] |
| | [1,2,4,3] after first loop iteration |
| | [1,4,2,3] after second loop iteration |
| | [4,1,2,3] after third loop iteration |

Prior to the first loop iteration, is the array partially correct in some way? Yes, if we restrict our attention to just the last element of the array, then we have $\text{is\_rotated}([4], [4])$. After the first iteration, is a bigger part of the array rotated? Yes, the last two elements of the array are rotated, that is, we have $\text{is\_rotated}([3,4], [4,3])$. Does this idea work for the second iteration? Yes, after the second iteration, the last three elements of the array are rotated, so we have $\text{is\_rotated}([2,3,4], [4,2,3])$. After the third and last iteration, we have $\text{is\_rotated}([1,2,3,4], [4,1,2,3])$. The loop invariant that we have just discovered is that the parts of array $A$ and $A\_orig$, starting at index $i$ and continuing to the end, satisfy the $\text{is\_rotated}$ predicate:

$\text{is\_rotated}(\text{copyOfRange}(A\_orig, i, A\_.\text{length}), \text{copyOfRange}(A, i, A\_.\text{length}))$

We use the $\text{copyOfRange}$ function from Java’s $\text{Arrays}$ class to create arrays for the regions of $A\_orig$ and $A$ from $i$ to the end.
assert $P$;
for (init; $C$; inc) {
    assert $R$ && $C$; // Condition: assert $P$; init; assert $R$;
    ...
    assert $S$; // Condition: assert $S$; inc; assert $R$
}  
assert $Q$; // Condition: $R$ && !$C$ implies $Q$

Figure 4: The Floyd-Hoare logic rule for for loops.

Armed with this loop invariant, we can fill in the details following the Floyd-Hoare logic rule for for loop, which is shown in Figure 4. The main idea behind how we reason about a loop is that instead of trying to reason about every iteration (which is impossible because we don’t know how many iterations will occur at runtime), we instead reason about one hypothetical iteration. We reason in a generic way, for example, not knowing the particular value of the loop index $i$, so that our reasoning is applicable to all the iterations. The rule in Figure 4 uses $P$ as a place-holder for what we know is true before the loop $Q$ for what we’d like to be true after the loop, and $R$ for the loop invariant. The initialization statement init executes once prior to the loop body, so we need to prove that, assuming $P$ is true, $R$ is true after executing init. The condition $C$ is also checked prior to entering the loop body, so we known that $C$ is true at the beginning of the loop body. By the end of the loop body, we will have shown that some predicate $S$ is true. To set up the next iteration of the loop, we must prove, assuming $S$ is true, that $R$ is true again after executing the inc statement. To finish things up, consider what happens at the end of the loop. The condition $C$ is false but the loop invariant $R$ is still true. So we must prove that $R$ and !$C$ implies $Q$.

Returning to where we left off in rotate_proof, let us fill in the assert statements according to the for loop rule, using the loop invariant we discovered above. We do not yet know what to fill in for the predicate that is true at the end of the loop body, predicate $S$.

assert equals (copyOfRange (A_orig, 0, A.length),
copyOfRange (A, 0, A.length))
    && 2 <= A.length;
for (int i = A.length - 1; i != 0; --i) {
    assert is_rotated (copyOfRange (A_orig, i, A.length),
copyOfRange (A, i, A.length))
    && i != 0;
    swap (A, i, i - 1);
    assert $S$;
}
assert is_rotated (A_orig, A);

As an aid to the proof, let us draw a picture of the hypothetical iteration that we shall reason about. We use the placeholders $w, x, y, z$ to represent the integers in those locations of the array. We know that the array $A$ has been rotated from index $i$ to the end, so we can see that $x, \ldots, y$ has been shifted to
the right by one position. We’ve written ? in the positions in A and A_orig that lower than i because the assert at the top of the loop body doesn’t say anything about those positions.

\[ A_{\text{orig}} = [..., ?, x, ..... y, z] \]
\[ A = [..., ?, z, x, ..... y] \]

The next statement in the loop body is the function call swap(A, i, i-1). If we go ahead and apply this operation to the current situation, we get the following situation which is rather unhelpful.

\[ A_{\text{orig}} = [..., ?, x, ..... y, z] \]
\[ A = [..., z, ?, x, ..... y] \]

We’d like to establish that A is now rotated from \( i-1 \) to the end, but we don’t know what element is in \( A[i] \) and \( A_{\text{orig}}[i-1] \) and whether they are equal to each other. (Remember, we must restrict our hypothetical reasoning to just what the assert says.) Of course, we know that A and A_orig start out equal to each other, and that prior to the current loop iterator, we haven’t changed anything from the beginning of the arrays to \( i-1 \). So we just need to add this information to our loop invariant. We add

\[ \text{equals}(\text{copyOfRange}(A_{\text{orig}}, 0, i), \text{copyOfRange}(A, 0, i)) \]

to the assert at the top of the loop body.

\[ \text{assert equals}(\text{copyOfRange}(A_{\text{orig}}, 0, A.\text{length}), \text{copyOfRange}(A, 0, A.\text{length})) \]
\[ \&\& 2 <= A.\text{length}; \]
\[ \text{for (int i = A.\text{length} - 1; i != 0; --i) { } } \]
\[ \text{assert is_rotated}(\text{copyOfRange}(A_{\text{orig}}, i, A.\text{length}), \text{copyOfRange}(A, i, A.\text{length})) \]
\[ \&\& \text{equals}(\text{copyOfRange}(A_{\text{orig}}, 0, i), \text{copyOfRange}(A, 0, i)) \]
\[ \&\& i != 0; \]
\[ \text{swap}(A, i, i-1); \]
\[ \text{assert S; } \]
\[ \text{assert is_rotated}(A_{\text{orig}}, A); \]

Now we can go back and update our picture of the hypothetical loop iteration. It now shows that A_orig and A contain the same elements, \( v, \ldots, w \), in the positions 0 through \( i-1 \).

\[ A_{\text{orig}} = [v, \ldots, w, x, \ldots, y, z] \]
\[ A = [v, \ldots, w, z, x, \ldots, y] \]

After swap(A, i, i-1), the w and z switch places in A. (We shall defer the proof of correctness of swap to the end of this note.)
So the array \( A \) is correctly rotated from index \( i - 1 \) to the end, and array \( A \) is the same as \( A_{\text{orig}} \) from the beginning through \( i - 2 \). We can use this as the predicate \( S \), filling in the assert at the end of the loop body:

```java
class rotate {
  assert equals (copyOfRange (A_orig, 0, A.length), copyOfRange (A, 0, A.length))
  && 2 <= A.length;
  for (int i = A.length - 1; i != 0; --i) {
    assert is_rotated (copyOfRange (A_orig, i, A.length),
    copyOfRange (A, i, A.length))
    && equals (copyOfRange (A_orig, 0, i),
    copyOfRange (A, 0, i))
    && i != 0;
    swap (A, i, i - 1);
    assert is_rotated (copyOfRange (A_orig, i - 1, A.length),
    copyOfRange (A, i - 1, A.length))
    && equals (copyOfRange (A_orig, 0, i - 1),
    copyOfRange (A, 0, i - 1));
  }
  assert is_rotated (A_orig, A);
}
```

With the \( S \) predicate specified, we can prove that the increment statement of the for loop takes us from \( S \) to \( R \):

```java
class rotate {
  assert is_rotated (copyOfRange (A_orig, i - 1, A.length),
  copyOfRange (A, i - 1, A.length))
  && equals (copyOfRange (A_orig, 0, i - 1),
  copyOfRange (A, 0, i - 1));
  --i;
  assert is_rotated (copyOfRange (A_orig, i, A.length),
  copyOfRange (A, i, A.length))
  && equals (copyOfRange (A_orig, 0, i),
  copyOfRange (A, 0, i))
}
```

This is trivial to verify because the only difference between the two assert statements is the change from \( i - 1 \) to \( i \).

Looking back at the rule for for statements (Figure 4), the last thing we need to verify is that \( R \) and not \( C \) implies \( Q \), that is,

- is_rotated(copyOfRange(A_orig, i, A.length), copyOfRange(A, i, A.length)), and
- \( i == 0 \)

implies is_rotated(A_orig, A). This is true because copyOfRange(A_orig, 0, A.length) is equal to \( A_{\text{orig}} \) and copyOfRange(A, 0, A.length) is equal to \( A \). This completes our proof of correctness of the rotate function. The completed rotate_proof function is shown in Figure ??.

Recall that the rotate function calls the swap function, which we define in Figure ???. We will not discuss the proof of correctness for swap in detail, but instead just give the version with assertions, swap_proof, in Figure ???. The reader is encouraged to check that the assertions constitute a proof of correctness.

Figure ?? shows the Floyd-Hoare logic rule for making a function call, in which we see a call from function \( g \) to another function \( f \). The idea is that if \( f \)
static void rotate_proof(int[] A) {
    int[] A_orig = copyOf(A, A.length);
    assert Arrays.equals(A_orig, A);
    if (2 <= A.length) {
        assert is_rotated(copyOfRange(A_orig, A.length - 1, A.length),
                         copyOfRange(A, A.length - 1, A.length))
        && Arrays.equals(copyOfRange(A_orig, 0, A.length),
                         copyOfRange(A, 0, A.length))
        && 2 <= A.length;
        for (int i = A.length - 1; i != 0; --i) {
            assert is_rotated(copyOfRange(A_orig, i, A.length),
                              copyOfRange(A, i, A.length))
            && Arrays.equals(copyOfRange(A_orig, 0, i),
                             copyOfRange(A, 0, i));
            swap(A, i, i - 1);
            assert is_rotated(copyOfRange(A_orig, i - 1, A.length),
                              copyOfRange(A, i - 1, A.length))
            && Arrays.equals(copyOfRange(A_orig, 0, i - 1),
                             copyOfRange(A, 0, i - 1));
        }
        assert is_rotated(A_orig, A);
    } else {
        assert Arrays.equals(A_orig, A) && ! (2 <= A.length);
        assert is_rotated(A_orig, A);
    }
    assert is_rotated(A_orig, A);
}

Figure 5: The rotate function with a proof of correctness via assertions that follow the rule of Floyd-Hoare logic.

static void swap(int[] A, int i, int j) {
    int tmp = A[i];
    A[i] = A[j];
    A[j] = tmp;
}

Figure 6: The swap function.

static void swap_proof(int[] A, int i, int j) {
    int[] A_orig = copyOf(A, A.length);
    int tmp = A[i];
    A[i] = A[j];
    A[j] = tmp;
}

Figure 7: The swap function.
starts with an assert (aka. precondition), then g must establish that precondition prior to the call. Note that the caller passes the arguments $a_1, \ldots, a_n$ for the parameters $p_1, \ldots, p_n$. So the caller must prove the precondition $P$ in terms of the arguments. Once the function call returns, the caller may assume that $Q$ is true, again for the arguments instead of the parameters.