Principled Programming
Introduction to Coding in Any Imperative Language

Tim Teitelbaum
Emeritus Professor
Department of Computer Science
Cornell University

One-Dimensional Array Rearrangements
The need to rearrange values in an array is commonplace, and facility in doing so is important.

Everyday experience is helpful, e.g., manipulating a hand of playing cards. However, beware that when cards are deleted or inserted, others move over automagically. A better analogy is cards in boxes, but even this is flawed because values are *copied* from variables, not *pulled*, like cards.

We consider:

- Reverse
- LeftShift
- LeftRotate
- Partitioning
- Collation
Application: Reverse the order of an array.
Application: Reverse the order of an array.

There is no shame in reasoning with concrete examples.
There is no shame in reasoning with concrete examples.
Application: Reverse the order of an array.

☞ There is no shame in reasoning with concrete examples.
Application: Reverse the order of an array.

There is no shame in reasoning with concrete examples.
Application: Reverse the order of (a subsequence of) an array.

/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R]  
in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
} /* Reverse */

☞ A header-comment says exactly what a method must accomplish, not how it does so.
Application: Reverse the order of (a subsequence of) an array.

/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R] in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
    while ( _______ ) {
        /* Swap A[L] and A[R]. */
        L++; R--;
    }
} /* Reverse */
/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R] in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
    while ( _______ ) {
        /* Swap A[L] and A[R]. */
        L++; R--;
    }
} /* Reverse */

Application: Reverse the order of (a subsequence of) an array.
Application: Reverse the order of (a subsequence of) an array.

```c
/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R]
in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
    while ( L<R ) {
        /* Swap A[L] and A[R]. */
        L++; R--;
    }
} /* Reverse */
```
Application: Reverse the order of (a subsequence of) an array.

```c
/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R] in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
    while ( L<R ) {
        /* Swap A[L] and A[R]. */
        L++; R--;
    }
} /* Reverse */
```

Validate output thoroughly.
Application: Reverse the order of (a subsequence of) an array.

```c
/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R] in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
    while ( L<R ) {
        /* Swap A[L] and A[R]. */
        L++; R--;
    }
} /* Reverse */
```

☞ Validate output thoroughly.
/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R] in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
    while ( L<R ) {
        /* Swap A[L] and A[R]. */
        L++; R--;
    }
} /* Reverse */

Application: Reverse the order of (a subsequence of) an array.

Validate output thoroughly.
Application: Reverse the order of (a subsequence of) an array.

/* Given int array A[0..n-1], reverse the order of the subsequence A[L..R] in situ without affecting the rest of A. */
static void Reverse( int A[], int L, int R ) {
    while ( L<R ) {
        /* Swap A[L] and A[R]. */
        L++; R--;
    }
} /* Reverse */

Be alert to high-risk coding steps associated with binary choices.
New Application: Shift an array left k places.
Application: Shift an array left $k$ places.

There is no shame in reasoning with concrete examples.
Application: Shift an array left k places.

/* Given array A[0..n-1], and 0≤k, shift elements of A left k places. Values shifted off the left end of the array are lost. Values not overwritten remain as they were originally. */

static void LeftShiftK( int[] A, int n, int k ) {
    /* LeftShiftK */
Application: Shift an array left k places.

/* Given array A[0..n-1], and 0≤k, shift elements of A left k places. Values shifted off the left end of the array are lost. Values not overwritten remain as they were originally. */
static void LeftShiftK( int[] A, int n, int k ) {
    for (int j=0; _____; j++) A[j] = A[j+k];
} /* LeftShiftK */
Application: Shift an array left k places.

/* Given array A[0..n-1], and 0≤k, shift elements of A left k places. Values shifted off the left end of the array are lost. Values not overwritten remain as they were originally. */
static void LeftShiftK( int[] A, int n, int k ) {
    for (int j=0; j<n-k; j++) A[j] = A[j+k];
} /* LeftShiftK */
Application: Shift an array left \(k\) places.

/* Given array \(A[0..n-1]\), and \(0 \leq k\), shift elements of \(A\) left \(k\) places. Values shifted off the left end of the array are lost. Values not overwritten remain as they were originally. */

static void LeftShiftK( int[] A, int n, int k ) {
    for (int j=0; j<n-k; j++) A[j] = A[j+k];
} /* LeftShiftK */

Boundary conditions. Dead last, but don’t forget them.
Application: Shift an array left \( k \) places.

```c
/* Given array \( A[0..n-1] \), and \( 0 \leq k \), shift elements of \( A \) left \( k \) places.
Values shifted off the left end of the array are lost. Values not 
overwritten remain as they were originally. */
static void LeftShiftK( int[] A, int n, int k ) {
    if ( k>0 )
        for (int j=0; j<n-k; j++) A[j] = A[j+k];
} /* LeftShiftK */
```

Boundary conditions. Dead last, but don’t forget them.

It would have been correct without this test, but offensive that for \( k==0 \) we would do the most work.
New application: Rotate an array left 1 place.

/* Given int array A[0..n-1], shift A[1..n-1] 1 place left, with the value originally in A[0] reentering at right in A[n-1]. */
static void LeftRotateOne(int A[], int n) {
    int temp = A[0];
    LeftShiftK(A, n, 1);
    A[n-1] = temp;
} /* LeftRotateOne */
Application: Rotate an array left 1 place.

/* Given int array A[0..n-1], shift A[1..n-1] 1 place left, with the value originally in A[0] reentering at right in A[n-1]. */
static void LeftRotateOne(int A[], int n) {
    int temp = A[0];
    LeftShiftK(A, n, 1);
    A[n-1] = temp;
} /* LeftRotateOne */

```c
static void LeftRotateOne(int A[], int n) {
    int temp = A[0];
    LeftShiftK(A, n, 1);
    A[n-1] = temp;
} /* LeftRotateOne */
```

**Application:** Rotate an array left 1 place.
New Application: Rotate an array left $k$ places.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
**Application:** Rotate an array left k places.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
Application: Rotate an array left k places.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */

We shall consider four distinct approaches:
- Repeated Left-Rotate-1
- Swap Generalization
- Three Flips
- Juggle in Cycles
Approach 1: Repeated left rotation 1 place.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
for (int j=0; j<k; j++) LeftRotateOne(A, n);
Approach 2: $k$-wide generalization of swap.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
Approach 2: k-wide generalization of swap.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int temp[] = new int[k];
Approach 2: k-wide generalization of swap.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int temp[] = new int[k];
/* temp[0..k-1] = A[0..k-1]; */
Approach 2: k-wide generalization of swap.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int temp[] = new int[k];
/* temp[0..k-1] = A[0..k-1]; */
LeftShiftK(A, n, k);
Approach 2: k-wide generalization of swap.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int temp[] = new int[k];
/* temp[0..k-1] = A[0..k-1]; */
LeftShiftK(A, n, k);
/* A[____..n-1] = temp[0..k-1]; */

Defer challenging code for later; do the easy parts first.
**Approach 2:** k-wide generalization of swap.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */

```java
int temp[] = new int[k];
/* temp[0..k-1] = A[0..k-1]; */
for (int j=0; j<k; j++) temp[j] = A[j];
LeftShiftK(A, n, k);
/* A[___..n-1] = temp[0..k-1]; */
```
Approach 2: k-wide generalization of swap.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int temp[] = new int[k];
/* temp[0..k-1] = A[0..k-1]; */
    for (int j=0; j<k; j++) temp[j] = A[j];
LeftShiftK(A, n, k);
/* A[____..n-1] = temp[0..k-1]; */
    for (int j=0; j<k; j++) A[_____] = temp[j];

Avoid gratuitous differences in code. Reuse code patterns, if possible.
Approach 2: k-wide generalization of swap.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int temp[] = new int[k];
/* temp[0..k-1] = A[0..k-1]; */
    for (int j=0; j<k; j++) temp[j] = A[j];
LeftShiftK(A, n, k);
/* A[n-k..n-1] = temp[0..k-1]; */
    for (int j=0; j<k; j++) A[n-k+j] = temp[j];
Approach 3: Three Flips. Consider the two parts of the array.

/* Given int array A[0..n-1], and integer k, 0 ≤ k < n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
Approach 3: Represent the values in those parts as green and blue arrows.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
Approach 3: Reverse first k

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
Reverse(A, 0, k-1);
Approach 3: Reverse first k, then rest of elements

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
k places, with values originally in A[0..k-1] reentering at right. */
Reverse(A, 0, k-1);
Reverse(A, k, n-1);
Approach 3: Reverse first $k$, then rest of elements

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
Reverse($A$, 0, $k-1$);
Reverse($A$, $k$, $n-1$);
Approach 3: Reverse first k, then rest of elements, then all elements.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
Reverse(A, 0, k-1);
Reverse(A, k, n-1);
Reverse(A, 0, n-1);
Approach 3: Reverse first k, then rest of elements, then all elements.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
Reverse(A, 0, k-1);
Reverse(A, k, n-1);
Reverse(A, 0, n-1);
Approach 4: Juggle elements in a stride of $k$.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */

int $p = 0$;                     // Start at $A[0]$
Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;                // Start at A[0]
int temp = A[0];          // and make a hole there.
Approach 4: Juggle elements in a stride of $k$.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ k places, with values originally in $A[0..k-1]$ reentering at right. */
int p = 0;             // Start at $A[0]$
int temp = $A[0]$;      // and make a hole there.
while ( __________ ) {
}

Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;          // Start at A[0]
int temp = A[0];    // and make a hole there.
while ( __________ ) {
    p = p+k; // Advance to the new hole.
}
/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;             // Start at A[0]
int temp = A[0];       // and make a hole there.

while ( __________ ) {
    p = p+k;            // Advance to the new hole.
}
Approach 4: Juggle elements in a stride of $k$.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
int p = 0; // Start at $A[0]$
int temp = $A[0]$; // and make a hole there.
while ( __________ ) {
    $p = p+k;$ // Advance to the new hole.
}
Approach 4: Juggle elements in a stride of \( k \).

/* Given int array \( A[0..n-1] \), and integer \( k \), \( 0 \leq k < n \), left shift \( A[k..n-1] \) \( k \) places, with values originally in \( A[0..k-1] \) reentering at right. */

```c
int p = 0;             // Start at \( A[0] \)
int temp = A[0];       // and make a hole there.

while ( __________ ) {
    A[p] = A[(p+k)%n];  // Fill hole at \( p \), making a new hole.
    p = (p+k)%n;        // Advance to the new hole.
}
```
Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;              // Start at A[0]
int temp = A[0];        // and make a hole there.
while ( __________ ) {
p = (p+k)%n;          // Advance to the new hole.
}
Approach 4: Juggle elements in a stride of \( k \).

```c
/* Given int array A[0..n-1], and integer k, 0\leq k<n, left shift A[k..n-1] 
k places, with values originally in A[0..k-1] reentering at right. */
int p = 0; // Start at A[0]
int temp = A[0]; // and make a hole there.
while ( __________ ) {
    p = (p+k)%n; // Advance to the new hole.
}
/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;             // Start at A[0]
int temp = A[0];       // and make a hole there.
while ( __________ ) {
    p = (p+k)%n;        // Advance to the new hole.
}
Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;             // Start at A[0]
int temp = A[0];       // and make a hole there.
while ( __________ ) {
    p = (p+k)%n;        // Advance to the new hole.
}
Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;          // Start at A[0]
int temp = A[0];   // and make a hole there.
while ( __________ ) {
p = (p+k)%n;        // Advance to the new hole.
}
Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;             // Start at A[0]
int temp = A[0];       // and make a hole there.
while ((p+k)%n!=0) {   // Stop if p is about to be 0 again.
    p = (p+k)%n;        // Advance to the new hole.
}
Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */

int p = 0;                 // Start at A[0]
int temp = A[0];          // and make a hole there.
while ( (p+k)%n!=0 ) {   // Stop if p is about to be 0 again.
    p = (p+k)%n;          // Advance to the new hole.
}
A[p] = temp;             // Fill the last hole from temp.

Approach 4: Juggle elements in a stride of k.
Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0 ≤ k < n, left shift A[k..n-1] k places, with values originally in A[0..k-1] reentering at right. */
int p = 0;       // Start at A[0]
int temp = A[0]; // and make a hole there.
while ((p+k)%n!=0) { // Stop if p is about to be 0 again.
    p = (p+k)%n;      // Advance to the new hole.
}                     
A[p] = temp;        // Fill the last hole from temp.
Are we done?
Approach 4: Juggle elements in a stride of $k$.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
int $p = 0$; // Start at $A[0]$ 
int $temp = A[0]$; // and make a hole there.
while ( (p+k)$\%$n != 0 ) { // Stop if $p$ is about to be 0 again.
    $p = (p+k)$\%$n$; // Advance to the new hole.
}
$A[p] = temp$; // Fill the last hole from $temp$. 

☞ Beware of premature self-satisfaction. 

Are we done?
Approach 4: Juggle elements in a stride of k.

/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
k places, with values originally in A[0..k-1] reentering at right. */

int p = 0;                  // Start at A[0]
int temp = A[0];            // and make a hole there.
while ( (p+k)%n!=0 ) {      // Stop if p is about to be 0 again.
    p = (p+k)%n;           // Advance to the new hole.
}
A[p] = temp;                // Fill the last hole from temp.

 Validate output thoroughly.

Are we done?
/* Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
k places, with values originally in A[0..k-1] reentering at right. */

int p = 0;             // Start at A[0]
int temp = A[0];       // and make a hole there.
while ((p+k)%n!=0) { // Stop if p is about to be 0 again.
    p = (p+k)%n;        // Advance to the new hole.
}
A[p] = temp;           // Fill the last hole from temp.

 Validate output thoroughly.

Are we done? Hardly.
Approach 4: Juggle elements in a stride of $k$.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
int $p = 0$;          // Start at $A[0]$
int temp = $A[0]$;   // and make a hole there.
while ((p+k)%n!=0) { // Stop if $p$ is about to be 0 again.
    $p = (p+k)\%n$;     // Advance to the new hole.
}                     // Fill the last hole from $temp$. 

 Validate output thoroughly.

Are we done? Hardly.
Approach 4: Juggle elements in a stride of $k$.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */

```c
int p = 0;             // Start at $A[0]$
int temp = $A[0]$;       // and make a hole there.
while ( (p+k)%n! = 0 ) { // Stop if $p$ is about to be 0 again.
    p = (p+k)%n;        // Advance to the new hole.
}
$A[p]$ = temp;           // Fill the last hole from temp.
```

Validate output thoroughly.

Are we done? Hardly. It only works if $k$ and $n$ are relatively prime!
Approach 4: Juggle elements in a stride of $k$.

/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
int $p = 0$; // Start at $A[0]$
int $temp = A[0]$; // and make a hole there.
while ((($p+k)\%n)!=$0) { // Stop if $p$ is about to be $0$ again.
    $p = ((p+k)\%n)$; // Advance to the new hole.
}
$A[p] = temp$; // Fill the last hole from $temp$.

* Validate output thoroughly.

Are we done? Hardly. It only works if $k$ and $n$ are relatively prime! Now what?
Abandon, or learn that each of $A[0..\gcd(k,n)-1]$ begins a disjoint cycle.

**Approach 4:** Juggle elements in a stride of $k$.

```c
/* Given int array $A[0..n-1]$, and integer $k$, $0 \leq k < n$, left shift $A[k..n-1]$ $k$ places, with values originally in $A[0..k-1]$ reentering at right. */
int g = \gcd(n,k);
for (int j=0; j<g; j++) {
    int p = j;  // Start at $A[j]$
    int temp = A[p]; // and make a hole there.
    while ( (p+k)\%n!=j ) { // Stop if $p$ is about to be $j$ again.
        p = (p+k)\%n;  // Advance to the new hole.
    }
    A[p] = temp; // Fill the last hole from temp.
}
### Assessment:

<table>
<thead>
<tr>
<th>Version of Left-Rotate-k</th>
<th>#moves</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated Left-Rotate-1</td>
<td>$k \cdot n$</td>
<td>Each Left-Rotate-1 moves all $n$ elements. Done $k$ times.</td>
</tr>
<tr>
<td>Swap Generalization</td>
<td>$n+k$</td>
<td>The copies into and out from temp do $2 \cdot k$ moves, and the shift does $n-k$ moves.</td>
</tr>
<tr>
<td>Three Flips</td>
<td>$2 \cdot n$</td>
<td>Each element moves once during the 1st two reverses, and then again for the 3rd reverse.</td>
</tr>
<tr>
<td>Juggle in Cycles</td>
<td>$n+\gcd(n,k)$</td>
<td>Each element moves once, plus the first element of each of the $\gcd(n,k)$ cycles must first be saved in temp.</td>
</tr>
</tbody>
</table>

Worst #moves, by far. Easiest to understand.
Assessment:

<table>
<thead>
<tr>
<th>Version of Left-Rotate-k</th>
<th>#moves</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated Left-Rotate-1</td>
<td>$k \cdot n$</td>
<td>Each Left-Rotate-1 moves all $n$ elements. Done $k$ times.</td>
</tr>
<tr>
<td>Swap Generalization</td>
<td>$n+k$</td>
<td>The copies into and out from $\text{temp}$ do $2 \cdot k$ moves, and the shift does $n-k$ moves.</td>
</tr>
<tr>
<td>Three Flips</td>
<td>$2 \cdot n$</td>
<td>Each element moves once during the 1st two reverses, and then again for the 3rd reverse.</td>
</tr>
<tr>
<td>Juggle in Cycles</td>
<td>$n+\gcd(n,k)$</td>
<td>Each element moves once, plus the first element of each of the $\gcd(n,k)$ cycles must first be saved in $\text{temp}$.</td>
</tr>
</tbody>
</table>

Reasonable #moves but not *in situ*, i.e., requires extra space for $\text{temp}$. 
Assessment:

<table>
<thead>
<tr>
<th>Version of Left-Rotate-k</th>
<th>#moves</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated Left-Rotate-1</td>
<td>( k \cdot n )</td>
<td>Each Left-Rotate-1 moves all ( n ) elements. Done ( k ) times.</td>
</tr>
<tr>
<td>Swap Generalization</td>
<td>( n+k )</td>
<td>The copies into and out from temp do ( 2 \cdot k ) moves, and the shift does ( n-k ) moves.</td>
</tr>
<tr>
<td>Three Flips</td>
<td>( 2 \cdot n )</td>
<td>Each element moves once during the 1st two reverses, and then again for the 3rd reverse.</td>
</tr>
<tr>
<td>Juggle in Cycles</td>
<td>( n + \gcd(n,k) )</td>
<td>Each element moves once, plus the first element of each of the ( \gcd(n,k) ) cycles must first be saved in temp.</td>
</tr>
</tbody>
</table>

Reasonable #moves. Good locality.
**Assessment:**

<table>
<thead>
<tr>
<th>Version of Left-Rotate-k</th>
<th>#moves</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated Left-Rotate-1</td>
<td>$k \cdot n$</td>
<td>Each Left-Rotate-1 moves all $n$ elements. Done $k$ times.</td>
</tr>
<tr>
<td>Swap Generalization</td>
<td>$n+k$</td>
<td>The copies into and out from temp do $2 \cdot k$ moves, and the shift does $n-k$ moves.</td>
</tr>
<tr>
<td>Three Flips</td>
<td>$2 \cdot n$</td>
<td>Each element moves once during the 1st two reverses, and then again for the 3rd reverse.</td>
</tr>
<tr>
<td>Juggle in Cycles</td>
<td>$n+\gcd(n,k)$</td>
<td>Each element moves once, plus the first element of each of the $\gcd(n,k)$ cycles must first be saved in temp.</td>
</tr>
</tbody>
</table>

Hardest to understand. Poor locality.
**Assessment:**

<table>
<thead>
<tr>
<th>Version of Left-Rotate-k</th>
<th>#moves</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated Left-Rotate-1</td>
<td>$k \cdot n$</td>
<td>Each Left-Rotate-1 moves all $n$ elements. Done $k$ times.</td>
</tr>
<tr>
<td>Swap Generalization</td>
<td>$n+k$</td>
<td>The copies into and out from temp do $2 \cdot k$ moves, and the shift does $n-k$ moves.</td>
</tr>
<tr>
<td>Three Flips</td>
<td>$2 \cdot n$</td>
<td>Each element moves once during the 1st two reverses, and then again for the 3rd reverse.</td>
</tr>
<tr>
<td>Juggle in Cycles</td>
<td>$n+\text{gcd}(n,k)$</td>
<td>Each element moves once, plus the first element of each of the $\text{gcd}(n,k)$ cycles must first be saved in temp.</td>
</tr>
</tbody>
</table>

Personal favorite, and really elegant!
Application: Rearrange an array into all red, then all white, then all blue.
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */

A statement-comment says exactly what code must accomplish, not how it does so.
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
while ( ______ ) __________

☞ If you “smell a loop”, write it down.
**Application:** Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
while ( _____ ) __________

---

To get to POST iteratively, choose a weakened POST as INVariant.
**Application:** Rearrange an array into all red, then all white, then all blue.

Here are four choices:

- To get to POST iteratively, choose a weakened POST as INVARINTANT.
Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices: How shall we choose?

Symmetric, so discard one arbitrarily.
Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices: How shall we choose?

Left more intuitive, because the ? region seems more familiar, so discard right.
Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices: How shall we choose?

? region of top has only one degree of freedom, but bottom has two. Discard top.
Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices: How shall we choose?

This will be our IN Variant.
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */

int k = ____; int w = ____; int b = ____;
while ( ______ ) __________
Given array $A[0..n-1]$ consisting of only three values (red, white, and blue), rearrange $A$ into all red, then white, then blue. */
/* $A[0..w-1]$ red, $A[w..k-1]$ white, $A[k..n-1]$ blue, for $0 \leq w \leq k \leq b \leq n$. */

int $k =$ ____;
int $w =$ ____;
int $b =$ ____;

while ( _____ ) __________

**Application:** Rearrange an array into all red, then all white, then all blue.

A Case Analysis in the loop body is often needed for characterizing different ways in which to decrease the loop variant while maintaining the loop invariant.
Given array $A[0..n-1]$ consisting of only three values (red, white, and blue), rearrange $A$ into all red, then white, then blue. */
/* $A[0..w-1]$ red, $A[w..k-1]$ white, $A[b..n-1]$ blue, for $0 \leq w \leq k \leq b \leq n$. */

```c
int k = ____;
int w = ____;
int b = ____;

while ( _____ )
    if ( A[k] == _____ ) __________
    else if ( A[k] == _____ ) __________
    else __________
```

A Case Analysis in the loop body is often needed for characterizing different ways in which to **decrease the loop variant** while maintaining the loop invariant.
/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____;
int w = ____;
int b = ____;
while ( ______ )
    if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */  b--; }
    else if ( A[k] == ______  ) __________
    else __________

VARIANT: b-k

**Application:** Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____; int w = ____; int b = ____;
while ( ______ )
    if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */  b--; }
    else if ( A[k] == ______  ) __________
    else __________

A Case Analysis in the loop body is often needed for characterizing different ways in which to **decrease the loop variant while maintaining the loop invariant.**
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____; int w = ____; int b = ____;
while ( _____ )
    if ( A[k] == B ) { /* Swap A[b-1] and A[k]. */ b--; }  
    else if ( A[k] == _____ ) __________
    else __________

☞ A Case Analysis in the loop body is often needed for characterizing different ways in which to decrease the loop variant while maintaining the loop invariant.
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____; int w = ____; int b = ____;
while ( ______ )
    if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */ b--; }
    else if ( A[k]==R ) { /* Swap A[w] and A[k]. */ w++; k++; }
    else __________

A Case Analysis in the loop body is often needed for characterizing different ways in which to decrease the loop variant while maintaining the loop invariant.
VARIANT: b-k

Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____; int w = ____; int b = ____;
while ( )
    if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */  b--; }
    else if ( A[k]==R ) { /* Swap A[w] and A[k]. */  w++; k++; }
    else ___________

A Case Analysis in the loop body is often needed for characterizing different ways in which to decrease the loop variant while maintaining the loop invariant.
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____; int w = ____; int b = ____;
while ( ____ )
    if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */  b--; }  
else if ( A[k]==R ) { /* Swap A[w] and A[k]. */  w++; k++; }  
else /* A[k]==W */ k++;

A Case Analysis in the loop body is often needed for characterizing different ways in which to decrease the loop variant while maintaining the loop invariant.
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____; int w = ____; int b = ____;
while ( _____ )
    if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */  b--; }
    else if ( A[k]==R ) { /* Swap A[w] and A[k]. */  w++; k++; }
    else /* A[k]==W */ k++;

[Style note: A Case Analysis in the loop body is often needed for characterizing different ways in which to decrease the loop variant while maintaining the loop invariant.]
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = ____; int w = ____; int b = ____;
while ( k!=b )
    if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */  b--; }
    else if ( A[k]==R ) { /* Swap A[w] and A[k]. */  w++; k++; }
    else /* A[k]==W */ k++;

VARIANT: b-k
Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. /* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */

```
int k = 0; int w = 0; int b = n;
while ( k!=b )
  if ( A[k]==B ) { /* Swap A[b-1] and A[k]. */  b--; } 
  else if ( A[k]==R ) { /* Swap A[w] and A[k]. */  w++; k++; } 
  else /* A[k]==W */ k++;
```

Application: Rearrange an array into all red, then all white, then all blue.
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
int k = 0; int w = 0; int b = n;
while ( k!=b )
  if ( A[k]==B ) {
    /* Swap A[b-1] and A[k]. */
    b--;
  }
  else if ( A[k]==R ) {
    /* Swap A[w] and A[k]. */
    w++; k++;
  }
  else /* A[k]==W */ k++;
Application: Rearrange an array into all red, then all white, then all blue.

/* Given array A[0..n-1] consisting of only three values (red, white, and blue), rearrange A into all red, then white, then blue. */
/* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0\leq w \leq k \leq b \leq n. */
int k = 0; int w = 0; int b = n;
while ( k!=b )
  if ( A[k]==B ) {
    /* Swap A[b-1] and A[k]. */
    b--;
  }
  else if ( A[k]==R ) {
    /* Swap A[w] and A[k]. */
    w++; k++;
  }
  else /* A[k]==W */ k++;
Performance:

• Constant work per iteration.

• Variant reduced by 1 on each iteration.

• Thus, running time linear in \( n \).
**New Application:** Rearrange (a segment of) an array into \(<p, ==p, and >p.\)

```c
/* Rearrange A[L..R-1] into all <p, then all ==p, then all >p. */
static void Partition( int A[], int L, int R, int p ) {
    /* Partition */
```

---

*A header-comment says exactly what a method must accomplish, not how it does so.*
Application: Rearrange (a segment of) an array into <p, ==p, and >p.

/* Rearrange A[L..R-1] into all <p, then all ==p, then all >p. */
static void Partition( int A[], int L, int R, int p ) {
    ⟨body of Dutch National Flag problem⟩
} /* Partition */
Application: Rearrange (a segment of) an array into <p, ==p, and >p.

/* Rearrange A[L..R-1] into all <p, then all ==p, then all >p. */
static void Partition( int A[], int L, int R, int p ) {
    /* A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n. */
    int k = 0; int w = 0; int b = n;
    while ( k!=b )
        if ( A[k]==B ) {
            /* Swap A[b-1] and A[k]. */
            b--;
        }
        else if ( A[k]==R ) {
            /* Swap A[w] and A[k]. */
            w++; k++;
        }
        else /* A[k]==W */ k++;
} /* Partition */
Application: Rearrange (a segment of) an array into <p, ==p, and >p.

/* Rearrange A[L..R-1] into all <p, then all ==p, then all >p. */
static void Partition( int A[], int L, int R, int p ) {
    int k = L; int w = L; int b = R;
    while ( k!=b )
        if ( A[k]>p ) {
            /* Swap A[b-1] and A[k]. */
            b--;
        } else if ( A[k]<p ) {
            /* Swap A[w] and A[k]. */
            w++; k++;
        } else /* A[k]==p */ k++;
} /* Partition */
**Application:** Rearrange (a segment of) an array into <p, ==p, and >p.

What value of p would tend to create “<p” and “>p” regions of near equal size?

If A[ L . . R-1] are in random order, any of those values is equally good for p.


This is central to a Divide and Conquer approach to sorting called QuickSort.
New Application: Collate ordered arrays A and B into array C.
Application: Collate ordered arrays A and B into array C.
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>40</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>30</td>
<td>50</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Application:** Collate ordered arrays A and B into array C.
Application: Collate ordered arrays A and B into array C.
### Application

Collate ordered arrays A and B into array C.
### Application

Collate ordered arrays A and B into array C.
Application: Collate ordered arrays A and B into array C.
### Application

Collate ordered arrays A and B into array C.
Application: Collate ordered arrays A and B into array C.

Collation is central to a Divide and Conquer approach to sorting called MergeSort.
Application: Collate ordered arrays A and B into array C.

/* Given ordered arrays A and B of lengths na and nb, create ordered array C of length na+nb consisting of those values. */
int C[] = new int[na+nb]; // C[0..kc-1] is collation of // A[0..ka-1] and B[0..kb-1].
int ka = ___; int kb = ___; int kc = ___; // Indices in A, B, and C.
/* Copy values from A or B into C until one array is exhausted. */
/* Copy remaining values into C from the unexhausted array, A or B. */
/* Given ordered arrays A and B of lengths na and nb, create ordered array C of length na+nb consisting of those values. */

int C[] = new int[na+nb]; // C[0..kc-1] is collation of // A[0..ka-1] and B[0..kb-1].

int ka = ___; int kb = ___; int kc = ___; // Indices in A, B, and C.

/* Copy values from A or B into C until one array is exhausted. */

while ( ____________ )
    if ( __________ ) { __________________________ } 
    else { _______________________________ }

/* Copy remaining values into C from the unexhausted array, A or B. */
/* Given ordered arrays A and B of lengths na and nb, create ordered
array C of length na+nb consisting of those values. */
int C[] = new int[na+nb];              // C[0..kc-1] is collation of
// A[0..ka-1] and B[0..kb-1].
int ka = ___; int kb = ___; int kc = ___; // Indices in A, B, and C.
/* Copy values from A or B into C until one array is exhausted. */
while ( _______________ )
    if ( ___________ ) { C[kc] = A[ka]; ka++; kc++; }
    else                 { __________________________ }
/* Copy remaining values into C from the unexhausted array, A or B. */

Application: Collate ordered arrays A and B into array C.
Application: Collate ordered arrays A and B into array C.

/* Given ordered arrays A and B of lengths na and nb, create ordered array C of length na+nb consisting of those values. */
int C[] = new int[na+nb];          // C[0..kc-1] is collation of
                                      // A[0..ka-1] and B[0..kb-1].

int ka = ___; int kb = ___; int kc = ___; // Indices in A, B, and C.
/* Copy values from A or B into C until one array is exhausted. */
while ( _______________ )
    if ( _____________ ) { C[kc] = A[ka]; ka++; kc++; }
    else
        { C[kc] = B[kb]; kb++; kc++; }
/* Copy remaining values into C from the unexhausted array, A or B. */
/* Given ordered arrays A and B of lengths na and nb, create ordered array C of length na+nb consisting of those values. */

int C[] = new int[na+nb];              // C[0..kc-1] is collation of
// A[0..ka-1] and B[0..kb-1].

int ka = ___; int kb = ___; int kc = ___; // Indices in A, B, and C.

/* Copy values from A or B into C until one array is exhausted. */
while ( ______________ )
    else { C[kc] = B[kb]; kb++; kc++; }

/* Copy remaining values into C from the unexhausted array, A or B. */

Application: Collate ordered arrays A and B into array C.
Given ordered arrays A and B of lengths na and nb, create ordered array C of length na+nb consisting of those values.

```
int C[] = new int[na+nb];  // C[0..kc-1] is collation of A[0..ka-1] and B[0..kb-1].
```

```
int ka = ___; int kb = ___; int kc = ___; // Indices in A, B, and C.
```

```
while ( ka<na && kb<nb )
  else                { C[kc] = B[kb]; kb++; kc++; }
/* Copy remaining values into C from the unexhausted array, A or B. */
```

**Application:** Collate ordered arrays A and B into array C.
/* Given ordered arrays A and B of lengths na and nb, create ordered array C of length na+nb consisting of those values. */

int C[] = new int[na+nb];              // C[0..kc-1] is collation of A[0..ka-1] and B[0..kb-1].

int ka = 0;  int kb = 0;  int kc = 0 ; // Indices in A, B, and C.

/* Copy values from A or B into C until one array is exhausted. */
while ( ka<na && kb<nb )
    else                { C[kc] = B[kb]; kb++; kc++; }

/* Copy remaining values into C from the unexhausted array, A or B. */

Application: Collate ordered arrays A and B into array C.
Application: Collate ordered arrays A and B into array C.

/* Given ordered arrays A and B of lengths na and nb, create ordered array C of length na+nb consisting of those values. */
int C[] = new int[na+nb]; // C[0..kc-1] is collation of // A[0..ka-1] and B[0..kb-1].

int ka = 0; int kb = 0; int kc = 0; // Indices in A, B, and C.
/* Copy values from A or B into C until one array is exhausted. */
while ( ka<na && kb<nb )
    else { C[kc] = B[kb]; kb++; kc++; }
/* Copy remaining values into C from the unexhausted array, A or B. */
while ( ka<na ) { C[kc] = A[ka]; ka++; kc++; }
while ( kb<nb ) { C[kc] = B[kb]; kb++; kc++; }