Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

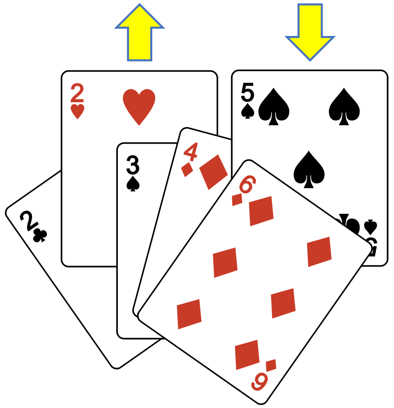
One-Dimensional Array Rearrangements

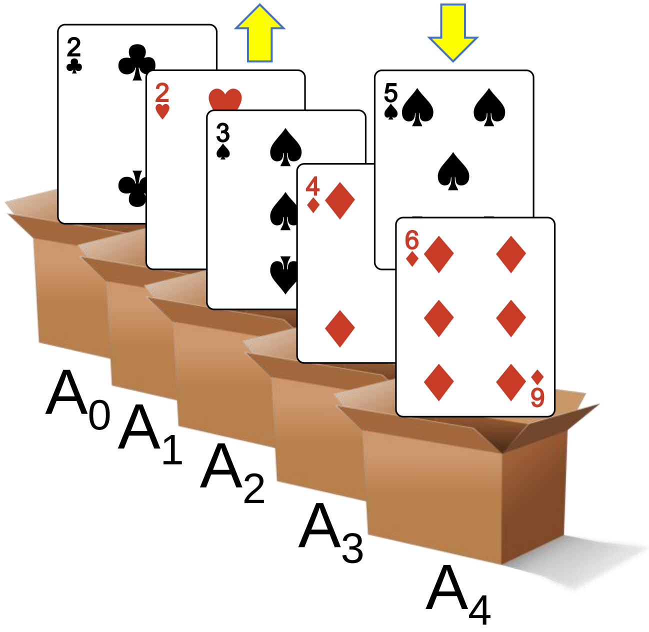
The need to rearrange values in an array is commonplace, and facility in doing so is important.

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.

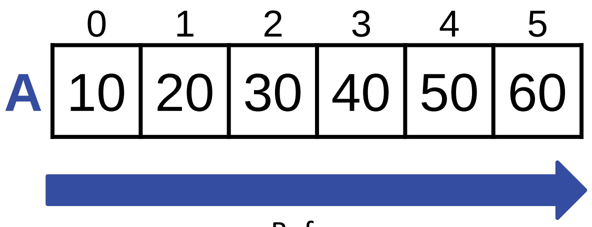
We consider:

- Reverse
- LeftShift
- LeftRotate
- Partitioning
- Collation





Reverse



Before

Application: Reverse the order of an array.

Reverse

	0	1	2	3	4	5
A	10	20	30	40	50	60

Before

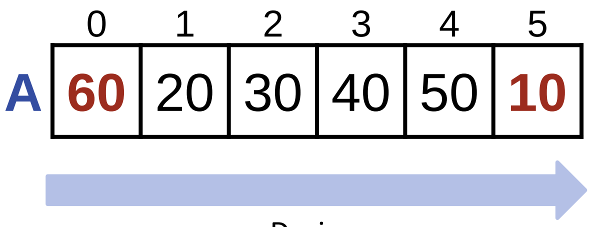
	0	1	2	3	4	5
A	60	50	40	30	20	10

After

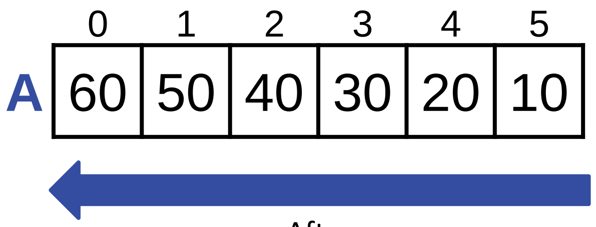
Application: Reverse the order of an array.

☞ **There is no shame in reasoning with concrete examples.**

Reverse



During

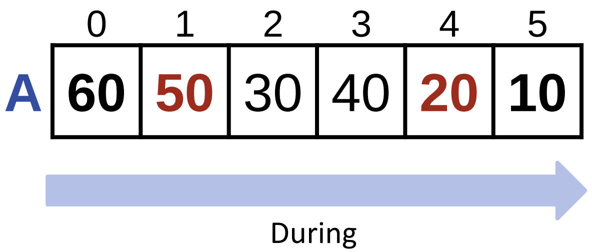


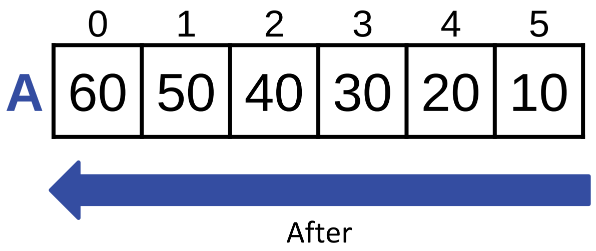
After

Application: Reverse the order of an array.

☞ **There is no shame in reasoning with concrete examples.**

Reverse

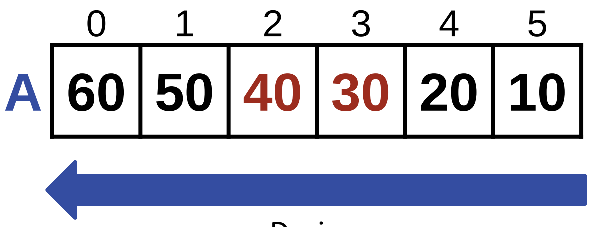




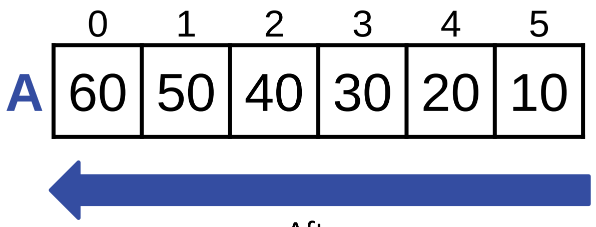
Application: Reverse the order of an array.

☞ **There is no shame in reasoning with concrete examples.**

Reverse



During

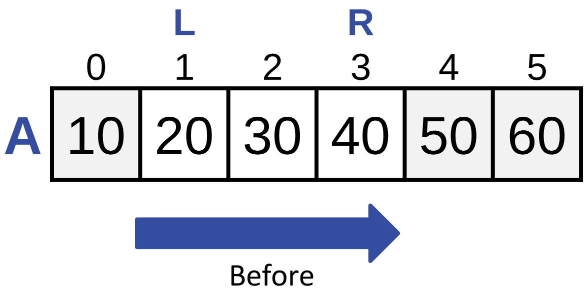


After

Application: Reverse the order of an array.

☞ **There is no shame in reasoning with concrete examples.**

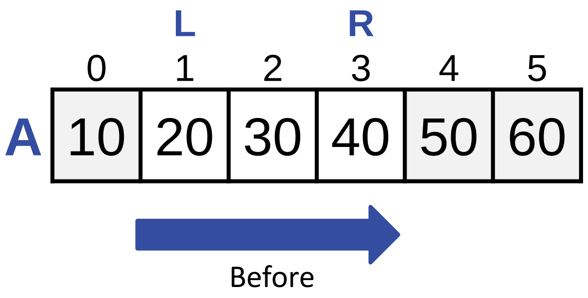
Reverse



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

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    ______
```

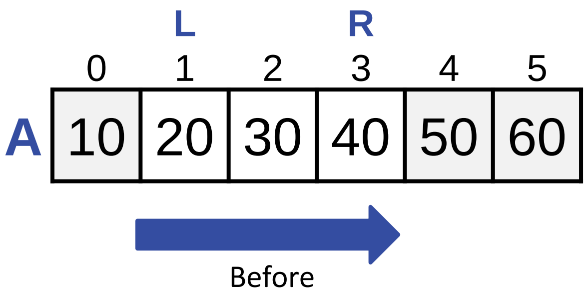
☞ **A header-comment says exactly what a method must accomplish, not how it does so.**



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

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while _______:

        ________
```

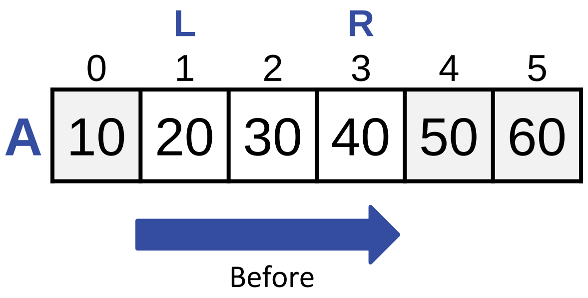
☞ **If you “smell a loop”, write it down.**



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

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while _______:
        #.Swap A[L] and A[R].
        L += 1;  R -= 1;
```

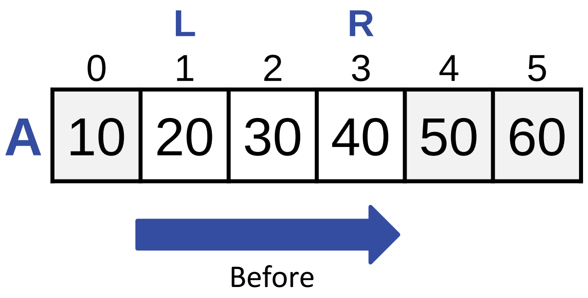
☞ **A statement-comment is written as a statement in a high-level language, e.g., English. As such, it is a specification for code not yet written.**



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

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while _______:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

☞ **Ignore fussy details for as long as possible.**

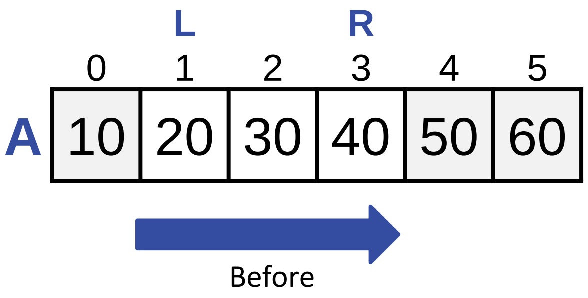


Application: Reverse the order of (a subsequence of) an array.

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while _______:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

But when the time comes, “you gotta do what you gotta do”.

☞ **Ignore fussy details for as long as possible.**

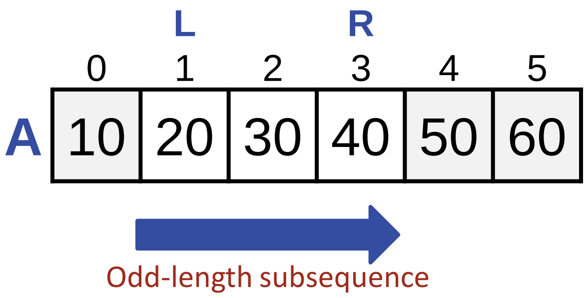


Application: Reverse the order of (a subsequence of) an array.

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while L < R:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

☞ **Be alert to high-risk coding steps associated with binary choices.**

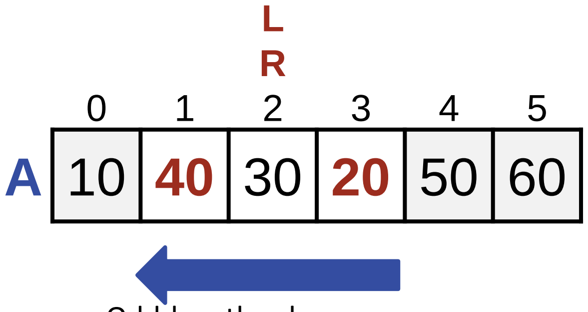
Reverse



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

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while L < R:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

☞ **Validate output thoroughly.**

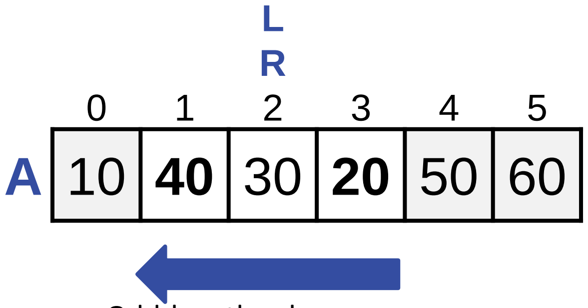


Application: Reverse the order of (a subsequence of) an array.

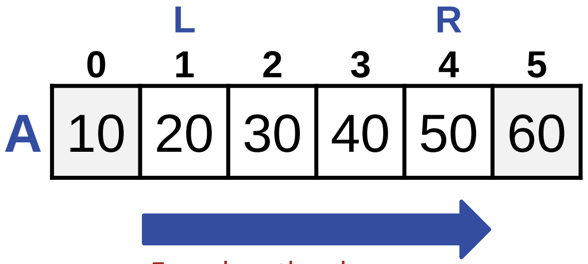
```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while L < R:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

☞ **Validate output thoroughly.**

Reverse



Odd-length subsequence

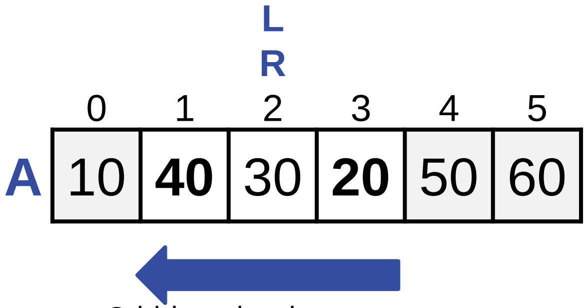


Even-length subsequence

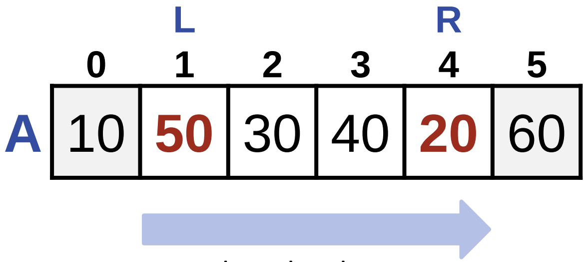
Application: Reverse the order of (a subsequence of) an array.

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while L < R:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

☞ **Validate output thoroughly.**



Odd-length subsequence

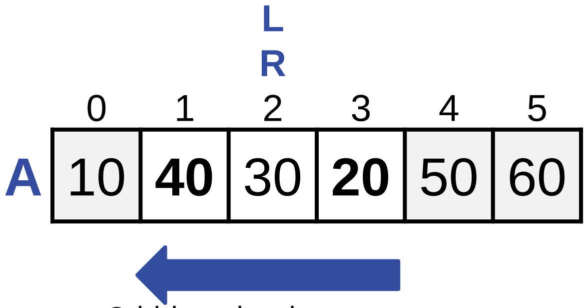


Even-length subsequence

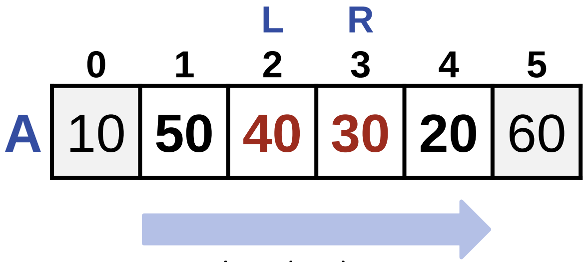
Application: Reverse the order of (a subsequence of) an array.

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while L < R:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

☞ **Validate output thoroughly.**



Odd-length subsequence



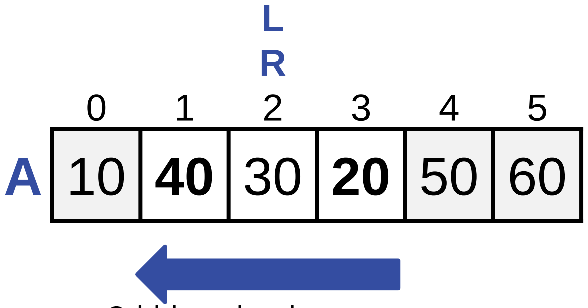
Even-length subsequence

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

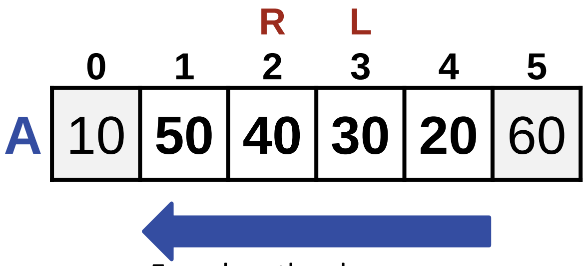
```python
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while L < R:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

☞ **Validate output thoroughly.**

Reverse



Odd-length subsequence



Even-length subsequence

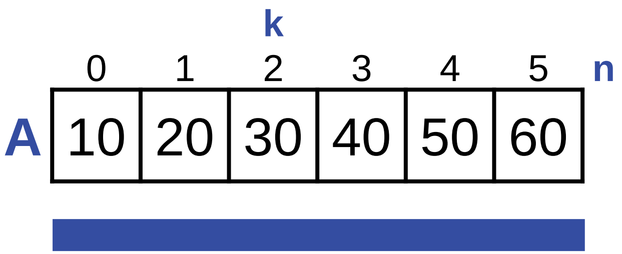
Application: Reverse the order of (a subsequence of) an array.

```
def reverse(A: list[int], L: int, R: int) -> None:
    """
    Given int array A[0..n-1], reverse(A,L,R) reverses the order of the
    subsequence A[L..R] in situ without affecting the rest of A.
    """

    while L < R:
        # Swap A[L] and A[R].
        temp = A[L];  A[L] = A[R];  A[R] = temp

        L += 1;  R -= 1
```

It would have been an error for the *condition* to have been `L!=R`.

☞ **Be alert to high-risk coding steps associated with binary choices.**

Left-Shift-k



New Application: Shift an array left k places.

Left-Shift-k

k

	0	1	2	3	4	5	n
A	10	20	30	40	50	60	

	0	1	2	3	4	5	n
A	**30**	**40**	**50**	**60**	50	60	

Application: Shift an array left k places.

☞ **There is no shame in reasoning with concrete examples.**

Application: Shift an array left k places.

```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """


    ________
```

☞ **A header-comment says exactly what a method must accomplish, not how it does so.**

Application: Shift an array left k places.

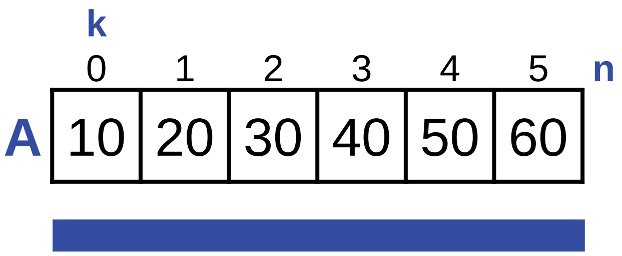
```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    for j in range(0, _____, 1): A[j] = A[j + k]
```

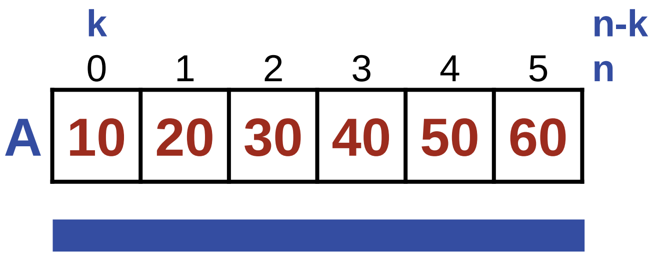
- ☞ **If you “smell a loop”, write it down.**
- ☞ **Decide first whether an iteration is indeterminate (use while) or determinate (use for).**

Left-Shift-k

Application: Shift an array left k places.

```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    for j in range(0, n - k, 1): A[j] = A[j + k]
```



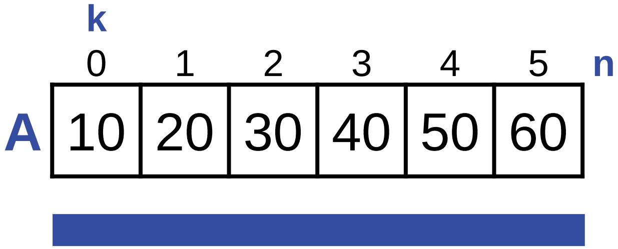


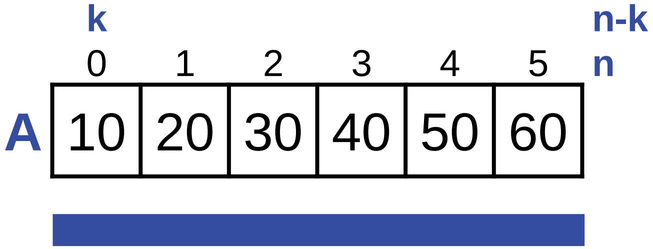
Application: Shift an array left k places.

```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    for j in range(0, n - k, 1): A[j] = A[j + k]
```

☞ **Boundary conditions. Dead last, but don't forget them.**

Left-Shift-k





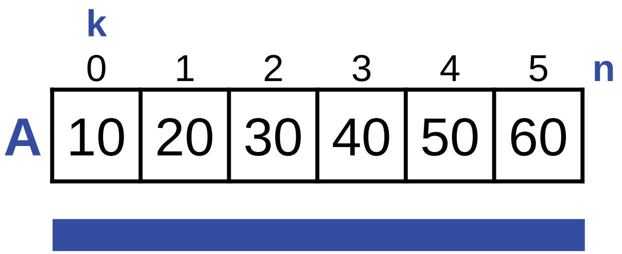
Application: Shift an array left k places.

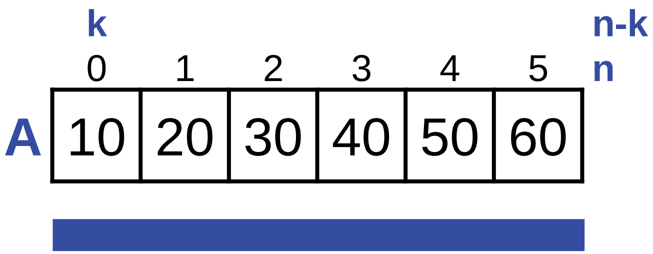
```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    if k > 0:
        for j in range(0, n - k, 1): A[j] = A[j + k]
```

> It would have been correct without this test, but offensive that for k==0 we would do the most work.

☞ **Boundary conditions. Dead last, but don't forget them.**

Left-Shift-k



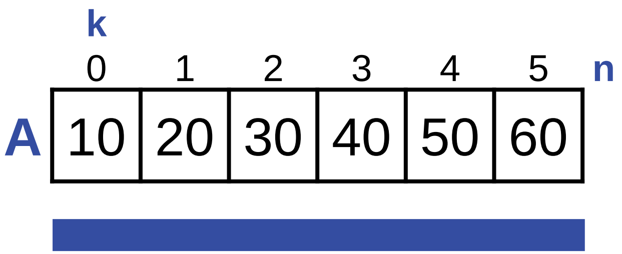


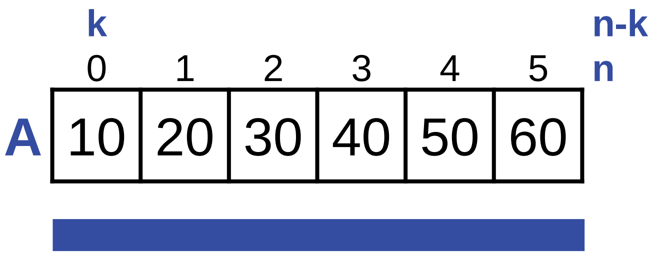
Application: Shift an array left k places.

```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    if k > 0:
        for j in range(0, n - k, 1): A[j] = A[j + k]
```

Default increment of `range()` is 1.

☞ **Eliminate clutter by using default values.**



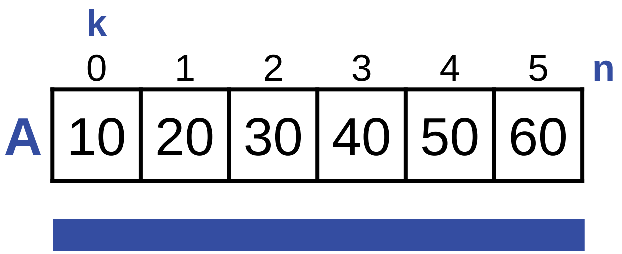


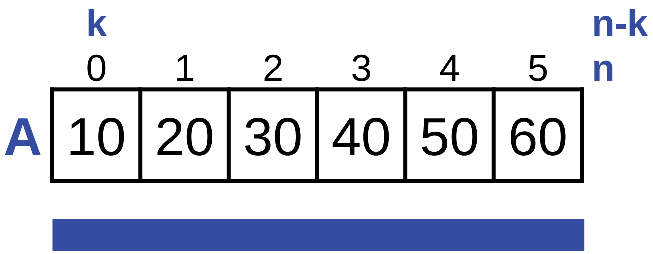
Application: Shift an array left k places.

```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    if k > 0:
        for j in range(0, n - k): A[j] = A[j + k]
```

☞ **Eliminate clutter by using default values.**

Left-Shift-k





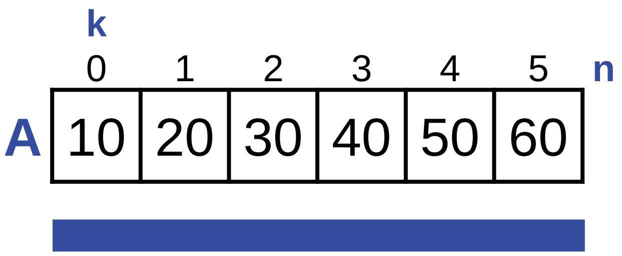
Application: Shift an array left k places.

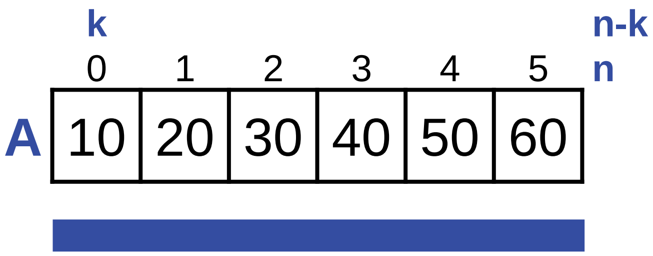
```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    if k > 0:
        for j in range(0, n - k): A[j] = A[j + k]
```

Default start of a `range()` is 0.

☞ **Eliminate clutter by using default values.**

Left-Shift-k



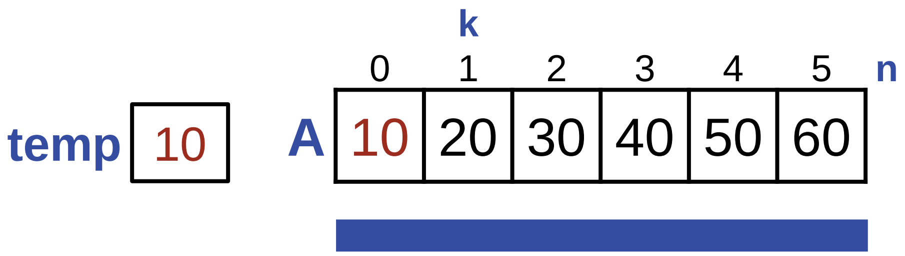


Application: Shift an array left k places.

```
def left_shift_k(A: list[int], n: int, k: int) -> None:
    """
    Given array A[0..n-1], and 0≤k, left_shift_k(A,n,k) shifts elements of A
    left k places. Values shifted off the left end of A are lost. Values not
    overwritten remain as they were originally.
    """

    if k > 0:
        for j in range(n - k): A[j] = A[j + k]
```

☞ **Eliminate clutter by using default values.**

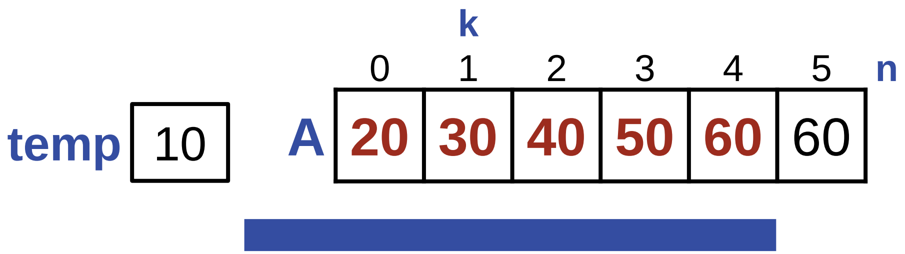
Left-Rotate-1



New application: Rotate an array left 1 place.

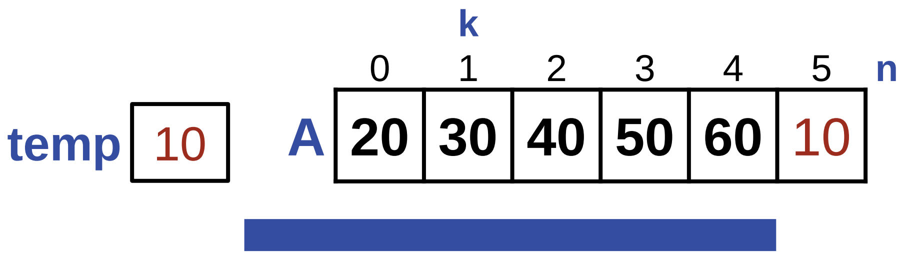
```
def left_rotate_one(A: list[int], n: int) -> None:
    """
    Given int array A[0..n-1], left_rotate_one(A,n) shifts A[1..n-1] left one
    place, with the value that was originally in A[0] reentering at the right
    in A[n-1].
    """

    temp = A[0]
    left_shift_k(A, n, 1)
    A[n - 1] = temp
```

Left-Rotate-1



Application: Rotate an array left 1 place.

```
def left_rotate_one(A: list[int], n: int) -> None:
    """
    Given int array A[0..n-1], left_rotate_one(A,n) shifts A[1..n-1] left one
    place, with the value that was originally in A[0] reentering at the right
    in A[n-1].
    """

    temp = A[0]
    left_shift_k(A, n, 1)
    A[n - 1] = temp
```



Application: Rotate an array left 1 place.

```
def left_rotate_one(A: list[int], n: int) -> None:
    """
    Given int array A[0..n-1], left_rotate_one(A,n) shifts A[1..n-1] left one
    place, with the value that was originally in A[0] reentering at the right
    in A[n-1].
    """

    temp = A[0]
    left_shift_k(A, n, 1)
    A[n - 1] = temp
```

Left-Rotate-k

New 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.
```

Left-Rotate-k

Left-Rotate-k

Application: Rotate an array left k places.

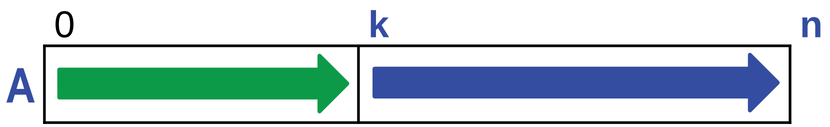
```
#.Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
```

We shall consider four distinct approaches:

- Repeated Left-Rotate-1
- Swap Generalization
- Three Flips
- Juggle in Cycles

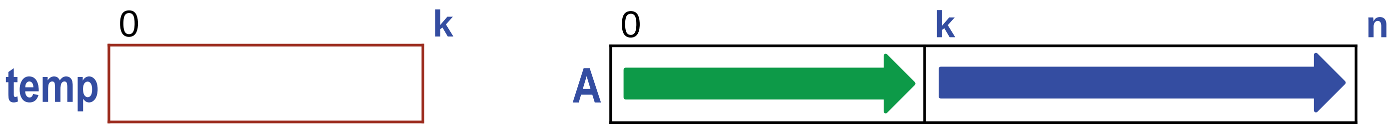
Approach 1: Repeated left rotation 1 place.

```
# 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 j in range(0, k): left_rotate_one(A, n)
```



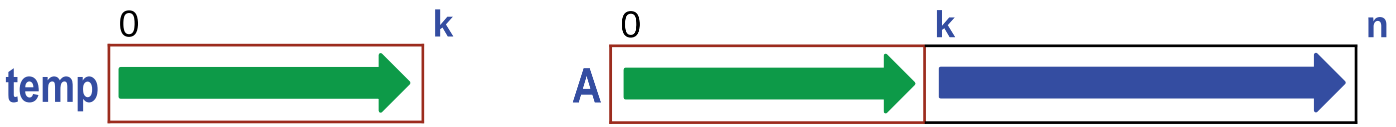
Approach 2: k-wide generalization of swap.

```
#.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.
```



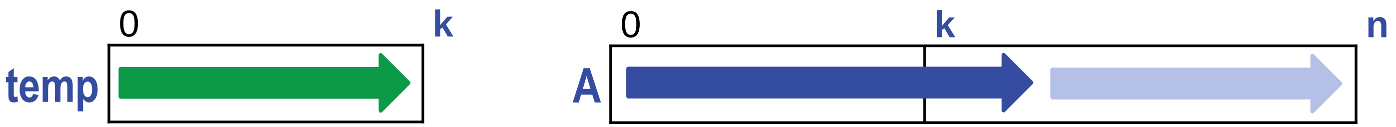
Approach 2: k-wide generalization of swap.

```
# 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.
temp = [0] * k
...
```



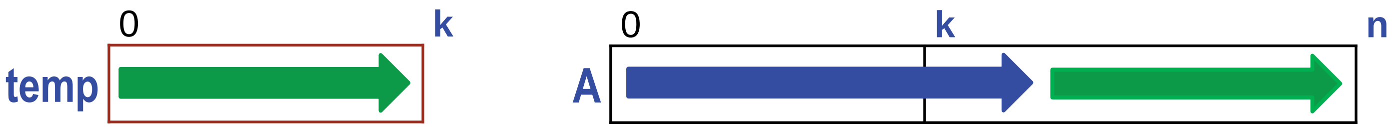
Approach 2: k-wide generalization of swap.

```
# 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.
# ---------------------------------------------------------------------
temp = [0] * 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.
# ------------------------------------------------------------------
temp = [0] * k
#.temp[0..k-1] = A[0..k-1]
left_shift_k(A, n, k)
...
```



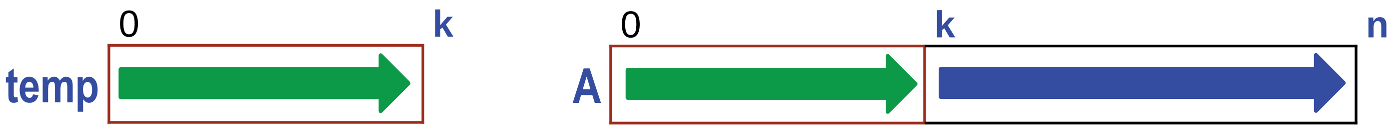
Approach 2: k-wide generalization of swap.

```
# 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.
# ---------------------------------------------------------------------
temp = [0] * k
#.temp[0..k-1] = A[0..k-1]
left_shift_k(A, n, k)
#.A[___..n-1] = temp[0..k-1]
```

☞ **Defer challenging code for later; do the easy parts first.**

Left-Rotate-k

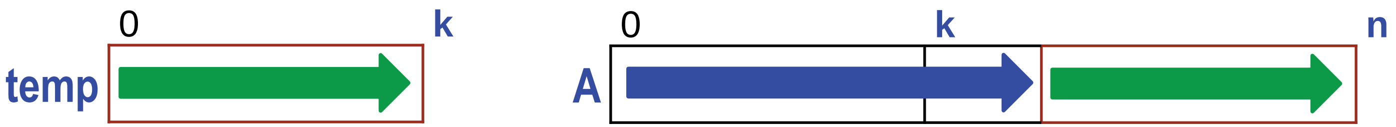
Swap Generalization



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.
# ----------------------------------------------------------------------
temp = [0] * k

# temp[0..k-1] = A[0..k-1]
for j in range(0, k): temp[j] = A[j]

left_shift_k(A, n, k)

#.A[___..n-1] = temp[0..k-1]
```

Left-Rotate-k
Swap Generalization



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.
# --------------------------------------------------------------------
temp = [0] * k

# temp[0..k-1] = A[0..k-1]
for j in range(0, k): temp[j] = A[j]

left_shift_k(A, n, k)

# A[___..n-1] = temp[0..k-1]
for j in range(0, k): A[_________] = temp[j]
```

☞ **Avoid gratuitous differences in code. Reuse code patterns, if possible.**

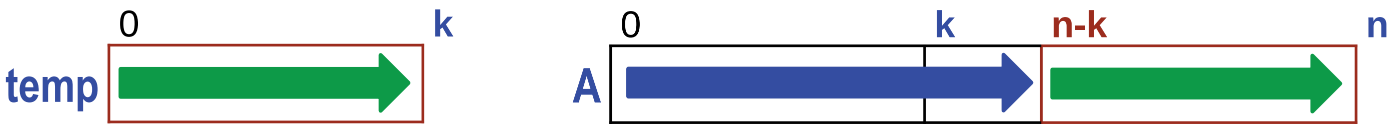
Left-Rotate-k

Swap Generalization

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.
# ---------------------------------------------------------------------
temp = [0] * k

# temp[0..k-1] = A[0..k-1]
for j in range(0, k): temp[j] = A[j]

left_shift_k(A, n, k)

# A[n-k..n-1] = temp[0..k-1]
for j in range(0, k): A[n – k + j] = temp[j]
```

Left-Rotate-k
Swap Generalization



Approach 2: k-wide generalization of swap.

Python has slices

```
# 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.
# ---------------------------------------------------------------------
temp = [0] * k

# temp[0..k-1] = A[0..k-1]
for j in range(0, k): temp[j] = A[j]

left_shift_k(A, n, k)

# A[n-k..n-1] = temp[0..k-1]
for j in range(0, k): A[n – k + j] = temp[j]
```

Approach 2: k-wide generalization of swap.

Python has slices

```
# 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.
# --------------------------------------------------------------------
temp = [0] * k

# temp[0..k-1] = A[0..k-1]
temp = A[:k]

left_shift_k(A, n, k)

# A[n-k..n-1] = temp[0..k-1]
A[(n – k):] = temp
```

Left-Rotate-k

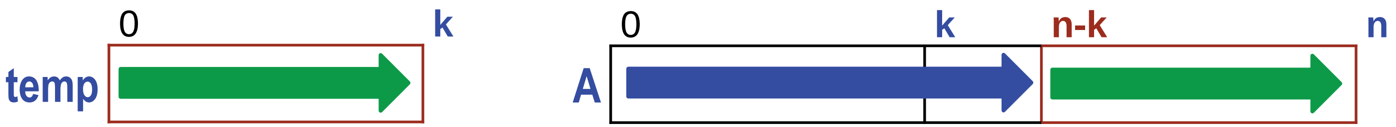
Swap Generalization

Approach 2: k-wide generalization of swap.

Python has slices

```
# 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.
temp = [0] * k
temp = A[:k]
left_shift_k(A, n, k)
A[(n – k):] = temp
```

☞ **Omit specifications whose implementations are at least as brief and clear as the specification itself.**



Approach 2: k-wide generalization of swap.

Python has slices

```
# 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.
temp = A[:k]
left_shift_k(A, n, k)
A[(n – k):] = temp
```

☞ **Omit unused assignments**

Approach 2: k-wide generalization of swap.

Python has slices

```
# 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.
# ---------------------------------------------------------------------
temp = A[:k]

# left_shift_k(A, n, k)
A[0:(n - k)] = A[k:]

A[(n – k):] = temp
```

Left-Rotate-k

Swap Generalization

Approach 2: k-wide generalization of swap.

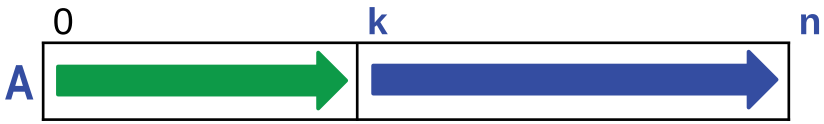
Python has slices

```
# 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.
temp = A[:k]
A[0:(n – k)] = A[k:]
A[(n – k):] = temp
```

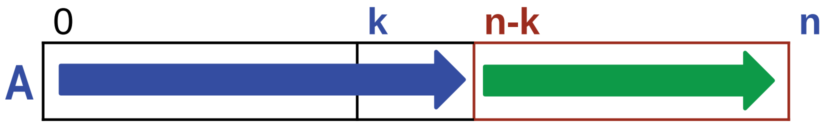
☞ **Omit specifications whose implementations are at least as brief and clear as the specification itself.**

Left-Rotate-k

Swap Generalization



before



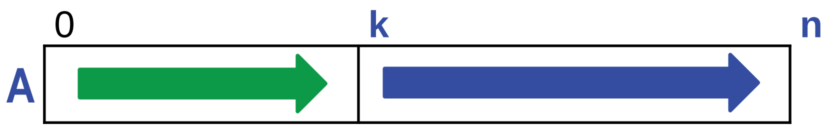
after

Approach 2: k-wide generalization of swap.

Python has concatenation

```
# 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.
A[0:n] = A[k:] + A[:k]
```

☞ **Use the expressive power of a language's operations.**



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.

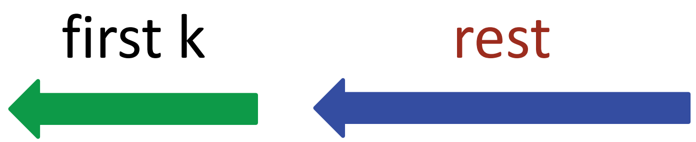
```
#.Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
```

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)
...
```

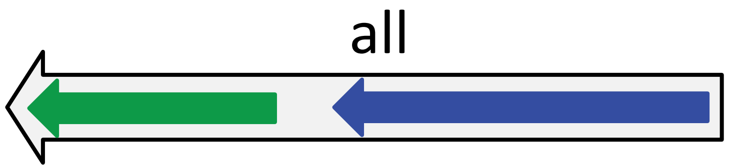
Left-Rotate-k

Three Flips



Approach 3: Reverse first k, then rest of elements

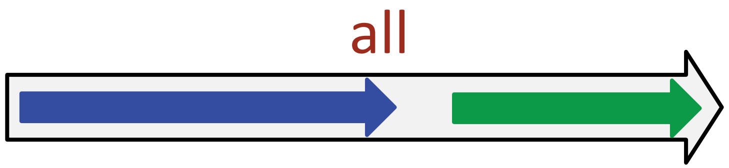
```
# Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
reverse(A, 0, k - 1)
reverse(A, k, n - 1)
...
```



Approach 3: Reverse first k, then rest of elements

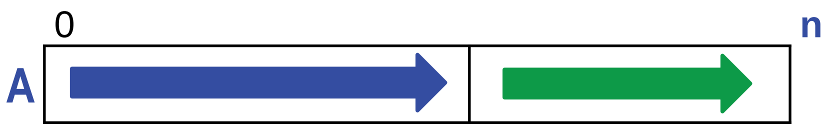
```
# Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
reverse(A, 0, k - 1)
reverse(A, k, n - 1)
...
```

Left-Rotate-k
Three Flips



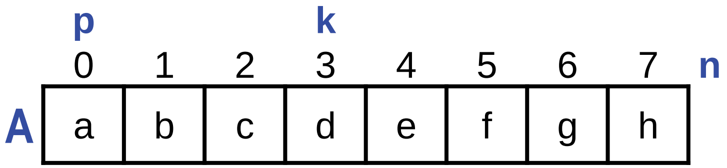
Approach 3: Reverse first k, then rest of elements, then all elements.

```
# 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)
```

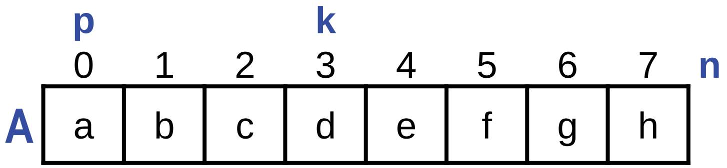


Approach 4: Juggle elements in a stride of k.

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

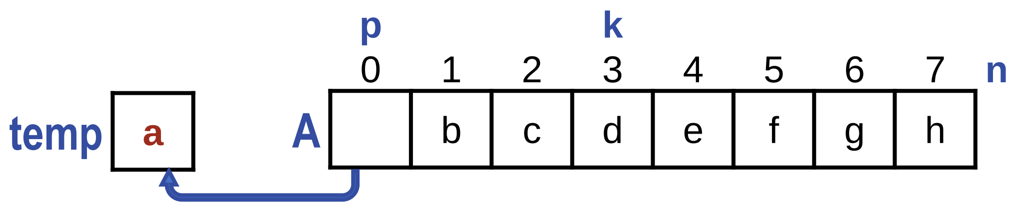
Left-Rotate-k

Juggle in Cycles



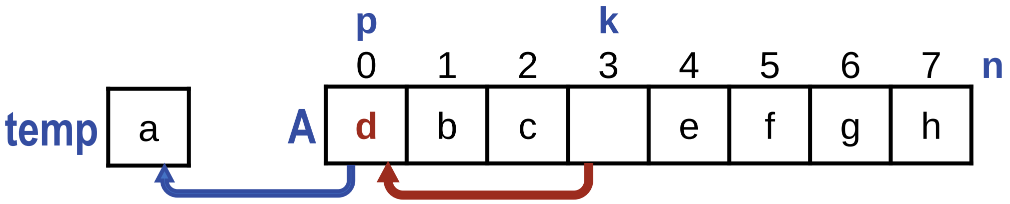
Approach 4: Juggle elements in a stride of k.

```
# 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.
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.
p = 0                              # Start at A[0]
temp = A[0]                        # and make a hole there.
...
```

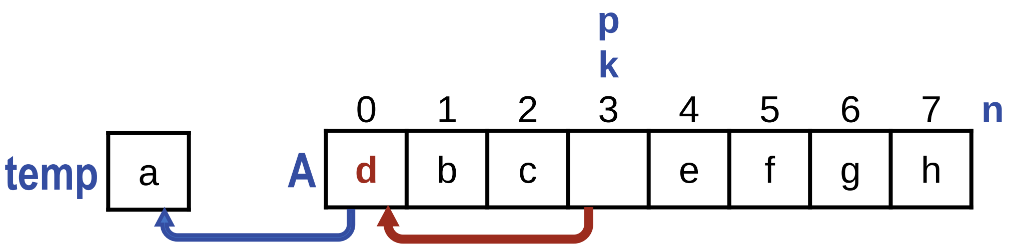


Approach 4: Juggle elements in a stride of k.

```
# Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
p = 0                             # Start at A[0]
temp = A[0]                       # and make a hole there.
while __________:
    A[p] = A[p + k]               # Fill hole at p, making a new hole.
    ...
```

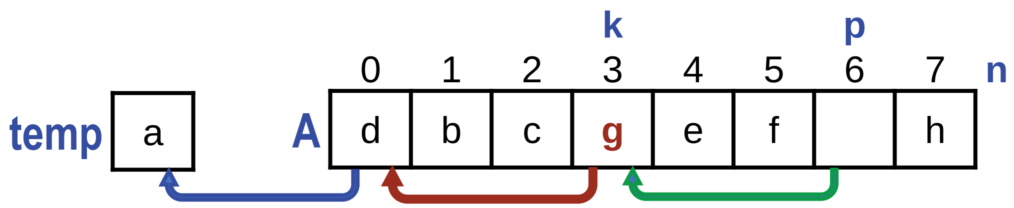
Left-Rotate-k

Juggle in Cycles



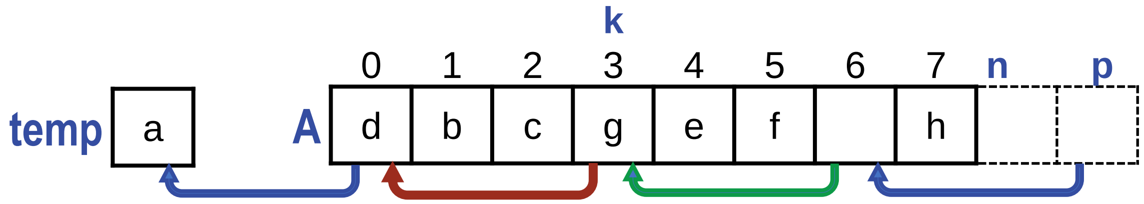
Approach 4: Juggle elements in a stride of k.

```
# 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.
p = 0                              # Start at A[0]
temp = A[0]                        # and make a hole there.
while __________:
    A[p] = A[p + k]                # Fill hole at p, making a new hole.
    p = p + k                      # Advance to the new hole.
...
```



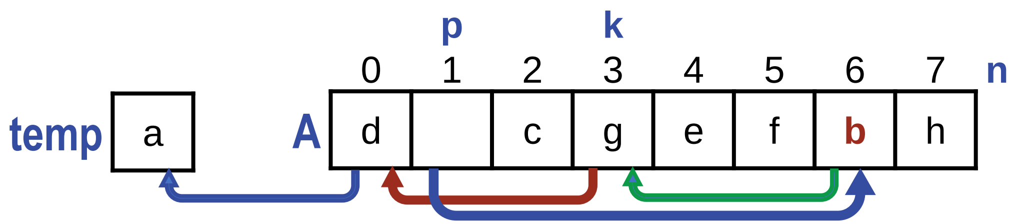
Approach 4: Juggle elements in a stride of k.

```
# Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
p = 0                             # Start at A[0]
temp = A[0]                       # and make a hole there.
while __________:
    A[p] = A[p + k]               # Fill hole at p, making a new hole.
    p = p + k                     # Advance to the new hole.
...
```



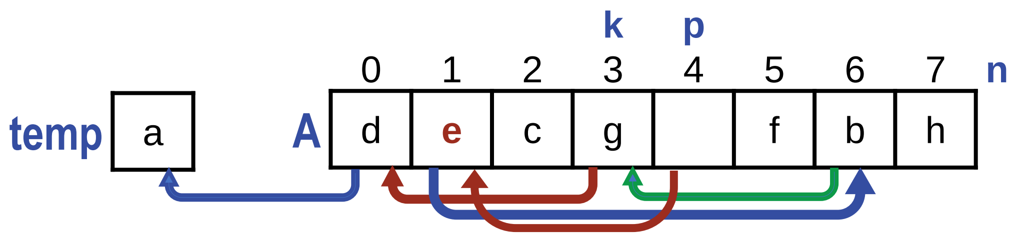
Approach 4: Juggle elements in a stride of k.

```
# Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
p = 0                             # Start at A[0]
temp = A[0]                       # and make a hole there.
while __________:
    A[p] = A[p + k]               # Fill hole at p, making a new hole.
    p = p + k                     # Advance to the new hole.
...
```



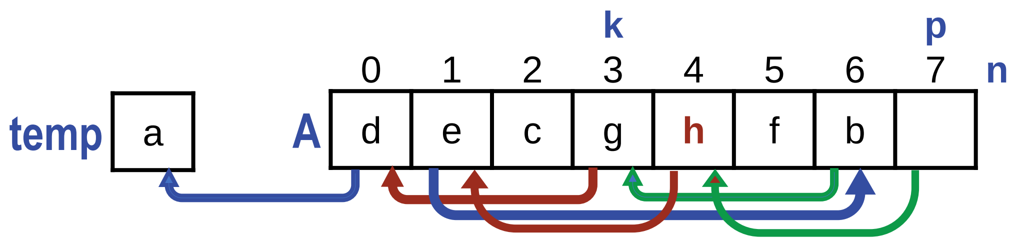
Approach 4: Juggle elements in a stride of k.

```
# Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
p = 0                             # Start at A[0]
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.
...
```



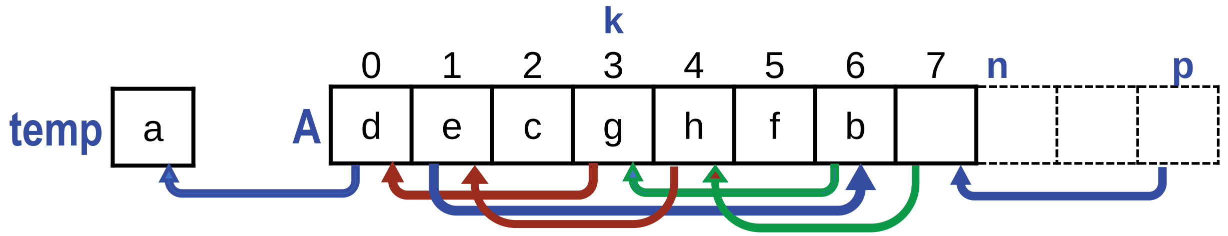
Approach 4: Juggle elements in a stride of k.

```
# 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.
p = 0                             # Start at A[0]
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.
...
```



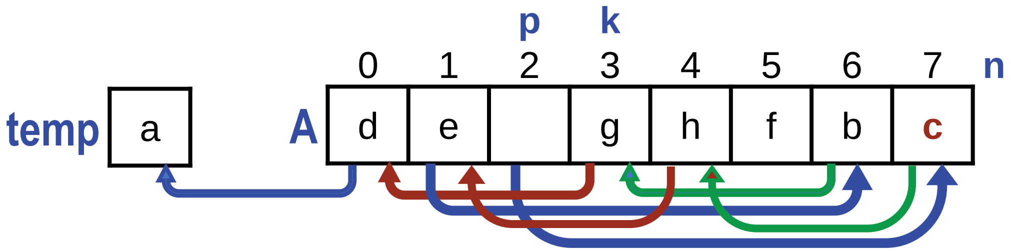
Approach 4: Juggle elements in a stride of k.

```
# 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.
p = 0                         # Start at A[0]
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.
...
```



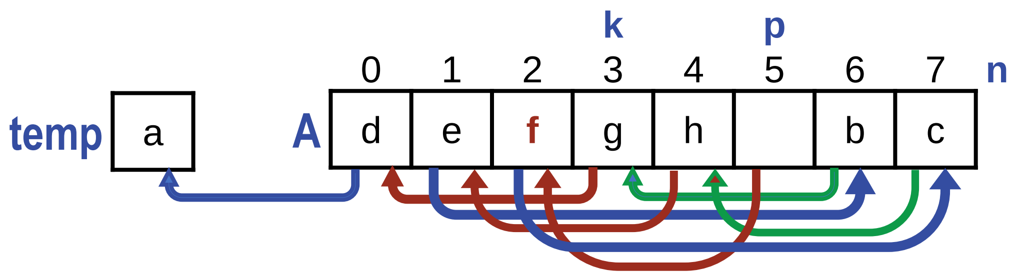
Approach 4: Juggle elements in a stride of k.

```
# 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.
p = 0                          # Start at A[0]
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.
...
```



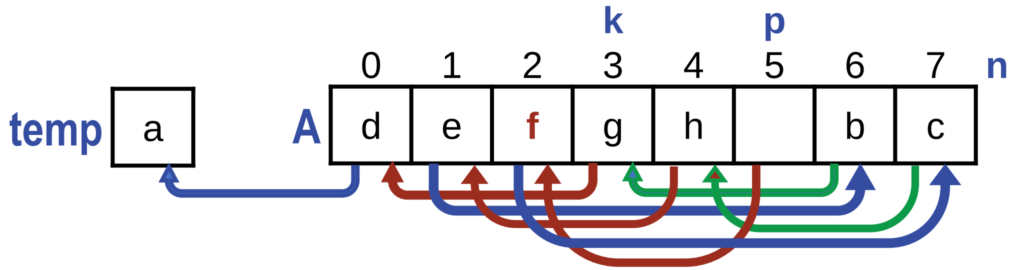
Approach 4: Juggle elements in a stride of k.

```
# 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.
p = 0                              # Start at A[0]
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.
...
```



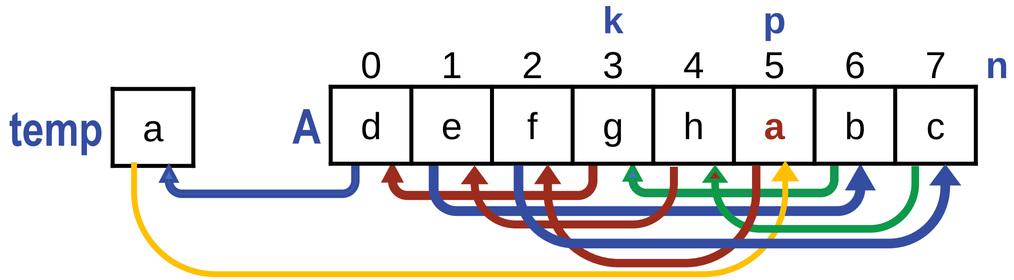
Approach 4: Juggle elements in a stride of k.

```
# 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.
p = 0                              # Start at A[0]
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.
p = 0                             # Start at A[0]
temp = A[0]                       # and make a hole there.
while ((p + k) % n) != 0:         # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]         # Fill hole at p, making a new hole.
    p = (p + k) % n               # Advance to the new hole.
...
```

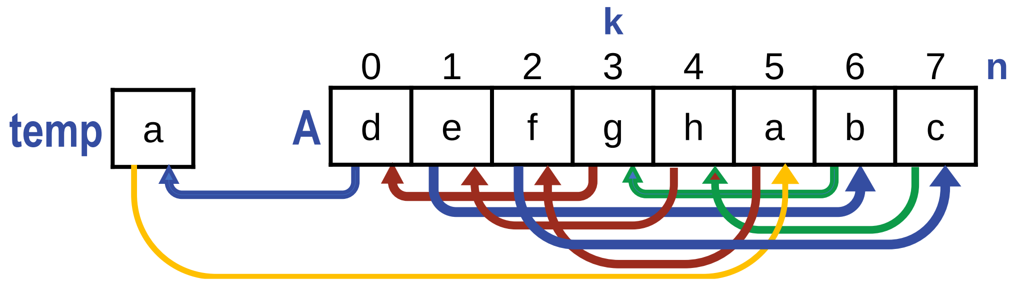


Approach 4: Juggle elements in a stride of k.

```
# 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.
p = 0                             # Start at A[0]
temp = A[0]                       # and make a hole there.
while ((p + k) % n) != 0:         # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]         # Fill hole at p, making a new hole.
    p = (p + k) % n               # Advance to the new hole.
A[p] = temp                       # Fill the last hole from temp.
```

Left-Rotate-k

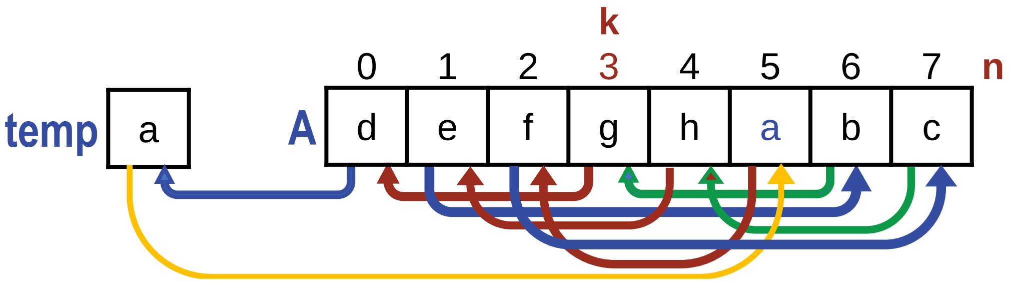
Juggle in Cycles



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.
p = 0                             # Start at A[0]
temp = A[0]                       # and make a hole there.
while ((p + k) % n) != 0:         # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]         # Fill hole at p, making a new hole.
    p = (p + k) % n               # Advance to the new hole.
A[p] = temp                       # Fill the last hole from temp.
```

Are we done?

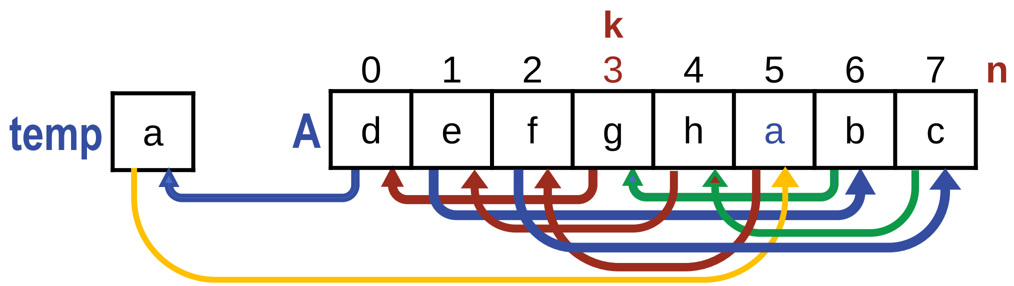


Approach 4: Juggle elements in a stride of k.

```
# Given int array A[0..n-1], and integer k, 0≤k<n, left shift A[k..n-1]
#   k places, with values originally in A[0..k-1] reentering at right.
p = 0                           # Start at A[0]
temp = A[0]                     # and make a hole there.
while ((p + k) % n) != 0:       # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]       # Fill hole at p, making a new hole.
    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?



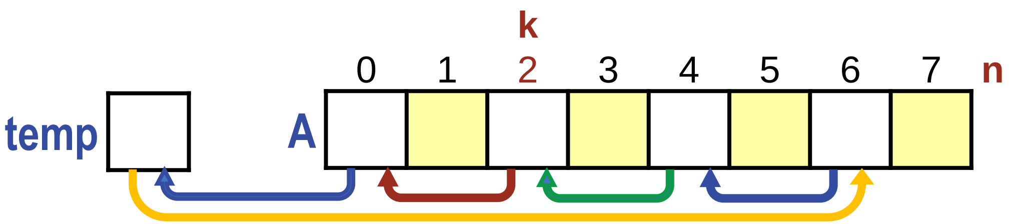
k	n
3	8

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.
p = 0                         # Start at A[0]
temp = A[0]                   # and make a hole there.
while ((p + k) % n) != 0:     # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]     # Fill hole at p, making a new hole.
    p = (p + k) % n           # Advance to the new hole.
A[p] = temp                   # Fill the last hole from temp.
```

☞ **Validate output thoroughly.**

Are we done?



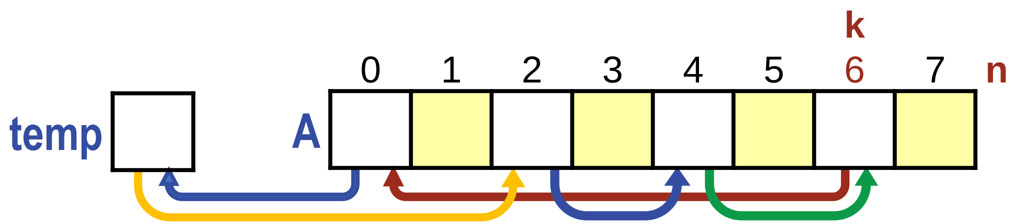
k	n	
3	8	✓
2	8	✗

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.
p = 0                         # Start at A[0]
temp = A[0]                   # and make a hole there.
while ((p + k) % n) != 0:     # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]     # Fill hole at p, making a new hole.
    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.



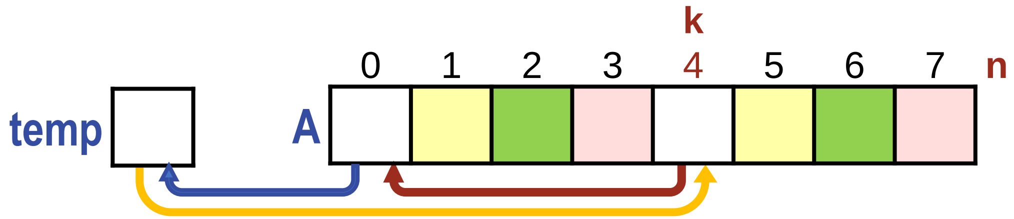
k	n	
3	8	✓
2	8	✗
6	8	✗

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.
p = 0                              # Start at A[0]
temp = A[0]                        # and make a hole there.
while ((p + k) % n) != 0:          # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]          # Fill hole at p, making a new hole.
    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.



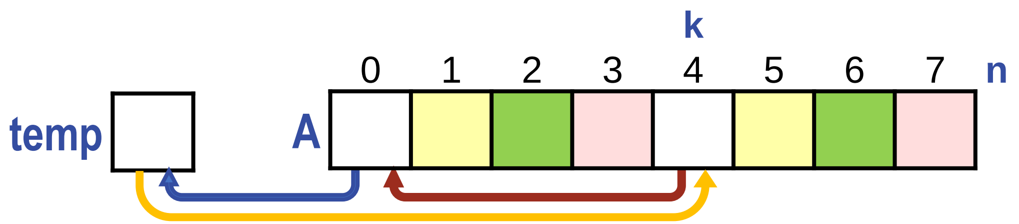
k	n	
3	8	✓
2	8	✗
6	8	✗
4	8	✗

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.
p = 0                          # Start at A[0]
temp = A[0]                    # and make a hole there.
while ((p + k) % n) != 0:      # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]      # Fill hole at p, making a new hole.
    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!



k	n	
3	8	✓
2	8	✗
6	8	✗
4	8	✗

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.
p = 0                              # Start at A[0]
temp = A[0]                        # and make a hole there.
while ((p + k) % n) != 0:          # Stop if p is about to be 0 again.
    A[p] = A[(p + k) % n]          # Fill hole at p, making a new hole.
    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.

```
# 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.
g = gcd(n, k);
for j in range(0, g):
    p = j                                # Start at A[j]
    temp = A[p]                          # and make a hole there.
    while ((p + k) % n) != 0:            # Stop if p is about to be j again.
        A[p] = A[(p + k) % n]            # Fill hole at p, making a new hole.
        p = (p + k) % n                  # Advance to the new hole.
    A[p] = temp                          # Fill the last hole from temp.
```

Assessment:

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

Worst #moves, by far. Easiest to understand.

Assessment:

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

Reasonable #moves but not *in situ*, i.e., requires extra space for `temp`.

Assessment:

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

Reasonable #moves. Good locality.

Assessment:

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

Hardest to understand. Poor locality.

Assessment:

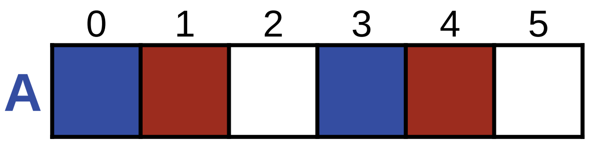
Version of Left-Rotate-k	#moves	Explanation
Repeated Left-Rotate-1	$k \cdot n$	Each Left-Rotate-1 moves all *n* elements. Done *k* times.
Swap Generalization	n+k	The copies into and out from `temp` do $2 \cdot k$ moves, and the shift does *n*-*k* moves.
✓ Three Flips	$2 \cdot n$	Each element moves once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

Personal favorite, and really elegant!

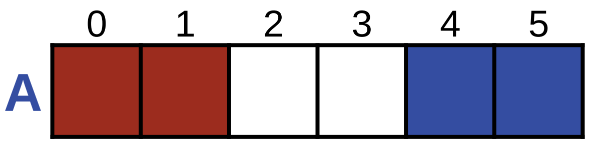
Dutch National Flag

New Application: The Dutch National Flag problem.

Dutch National Flag



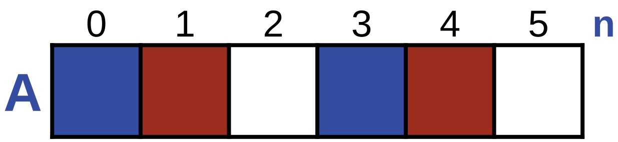
before



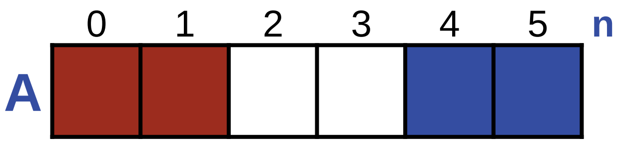
after

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

Dutch National Flag



before



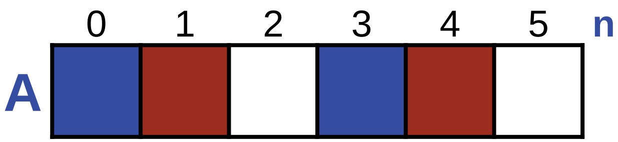
after

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

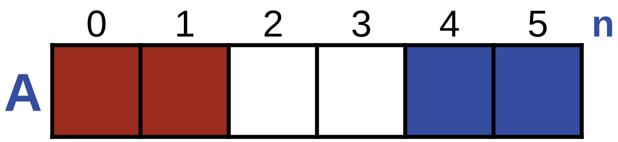
```
#.Given array A[0..n-1] consisting of only three values (red, white, and blue),
#   rearrange A into all red, then white, then blue.
```

☞ **A statement-comment says exactly what code must accomplish, not how it does so.**

Dutch National Flag



before



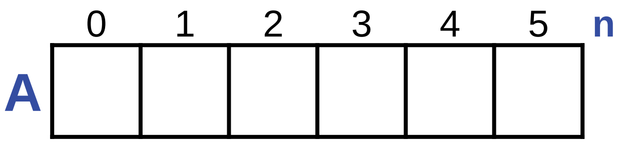
after

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

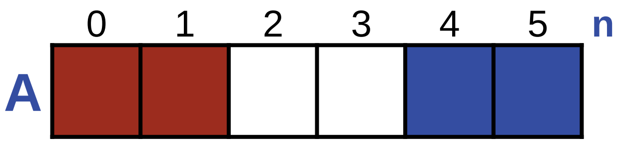
```
# Given array A[0..n-1] consisting of only three values (red, white, and blue),
#   rearrange A into all red, then white, then blue.
while ______: _________
```

☞ **If you “smell a loop”, write it down.**

Dutch National Flag



intermediate



after

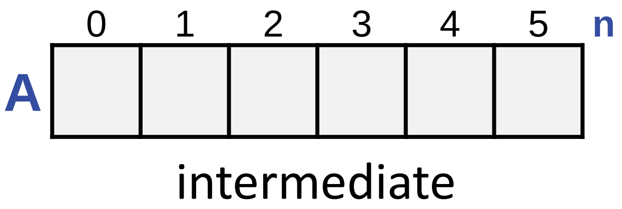
POST

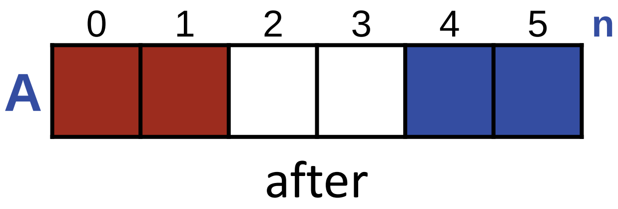
Application: Rearrange an array into all red, then all white, then all blue.

```
# 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.**

Dutch National Flag

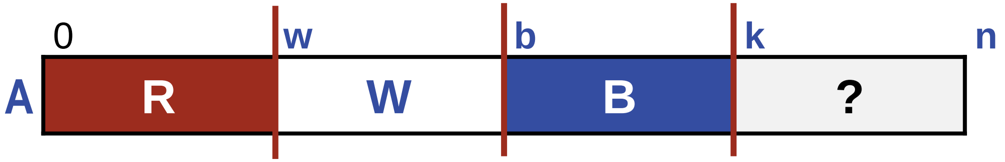


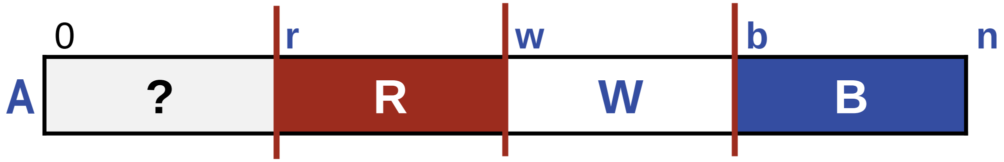


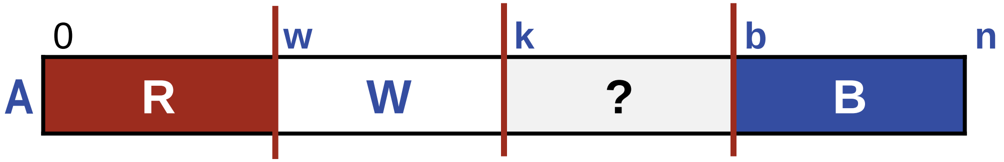
POST

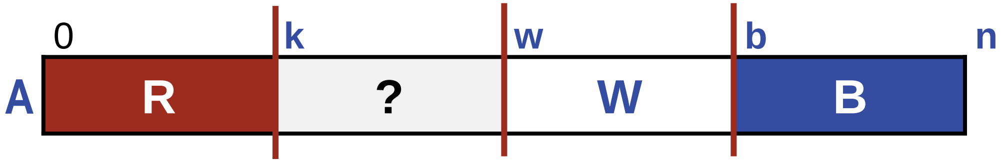
Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices for a weakened POST:

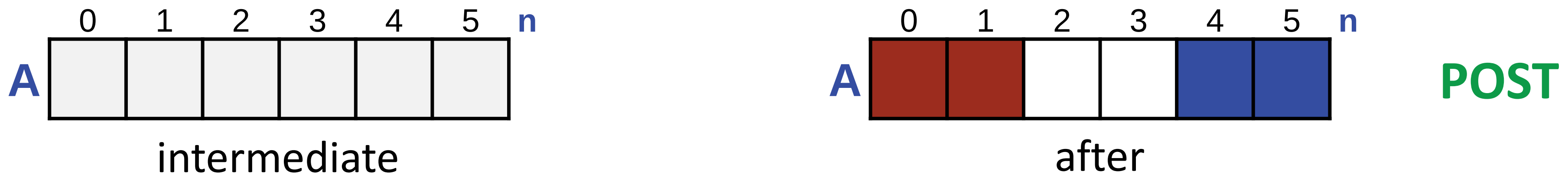






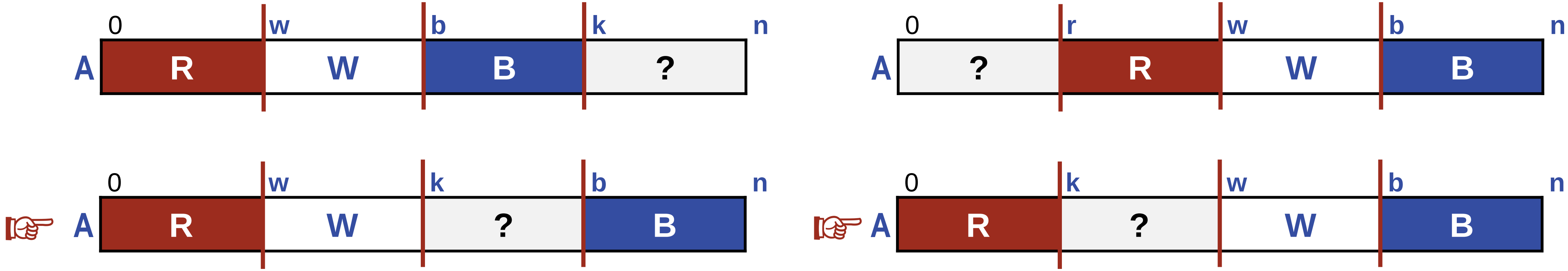


☞ **To get to POST iteratively, choose a weakened POST as INVARIANT.**



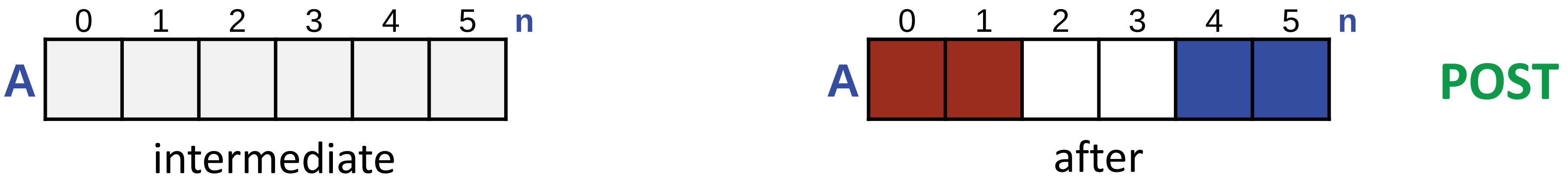
Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices for a weakened POST: How shall we choose?



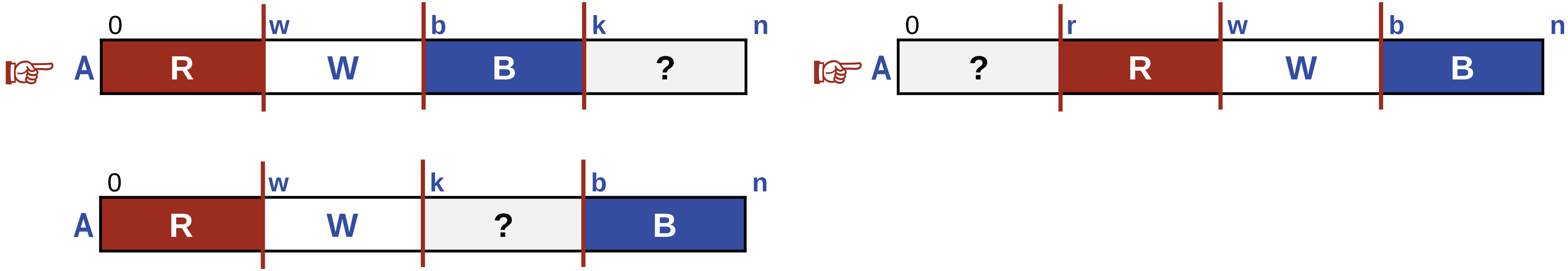
Symmetric, so discard one arbitrarily.

Dutch National Flag

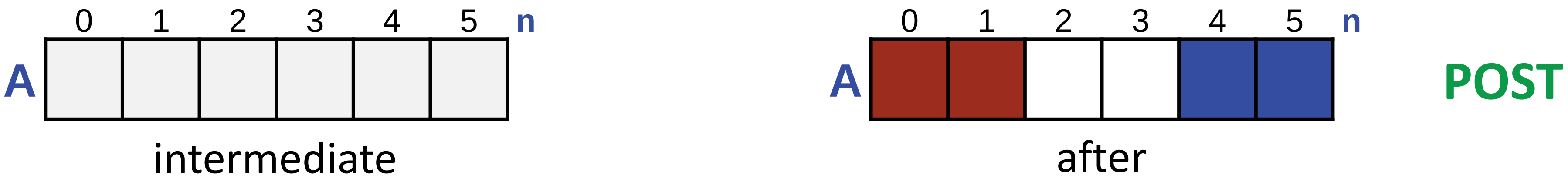


Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices for a weakened POST: How shall we choose?

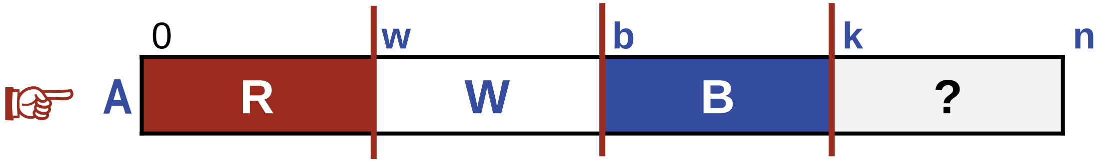


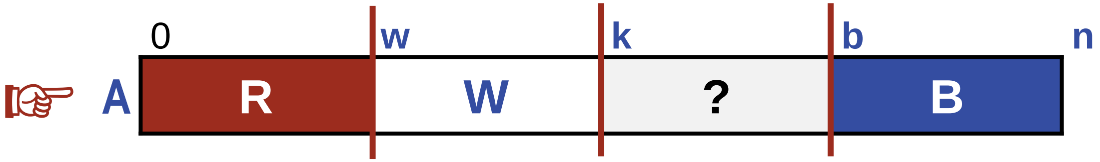
Left more intuitive, because the ? region seems more familiar, so discard right.



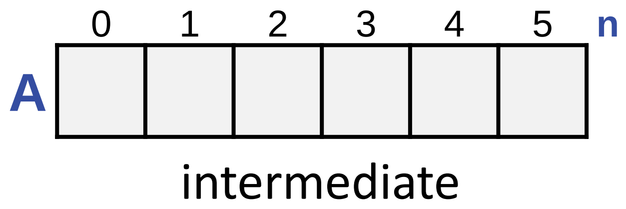
Application: Rearrange an array into all red, then all white, then all blue.

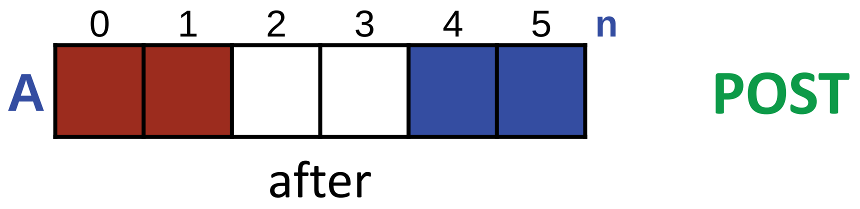
Here are four choices for a weakened POST: How shall we choose?





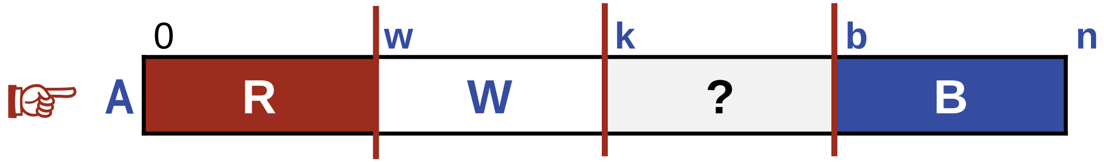
? region of top has only one degree of freedom, but bottom has two. Discard top.





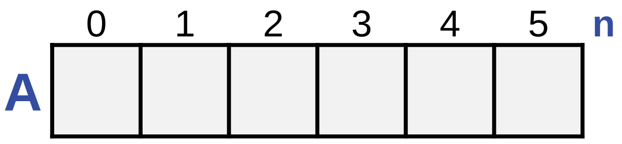
Application: Rearrange an array into all red, then all white, then all blue.

Here are four choices for a weakened POST: How shall we choose?

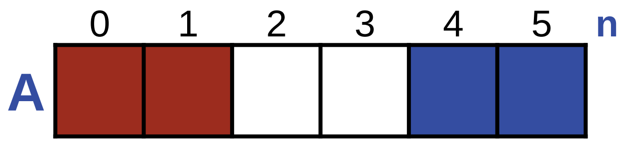


This will be our **INVARIANT**.

Dutch National Flag



intermediate

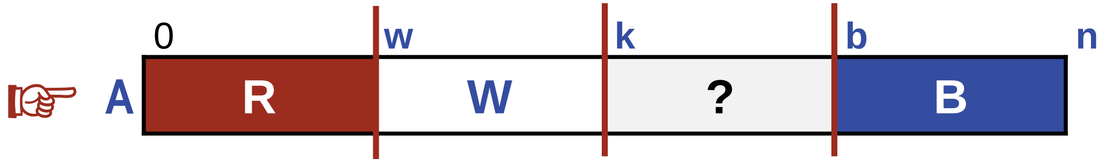


after

POST

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

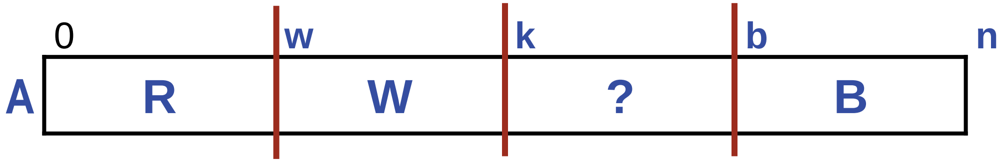
Here are four choices for a weakened POST: How shall we choose?



This will be our **INVARIANT**.

We have illustrated that program design and programming can be driven by consideration of the different possible invariants you can think of. One might call this "invariant-driven programming". In this mode of programming, the invariant comes first, not as an afterthought to justify a loop you have already written.

Dutch National Flag

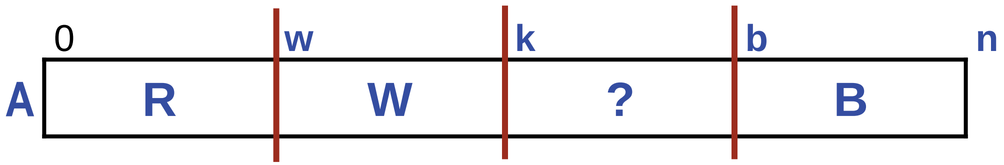


VARIANT: b-k

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.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______: _________
```

Dutch National Flag

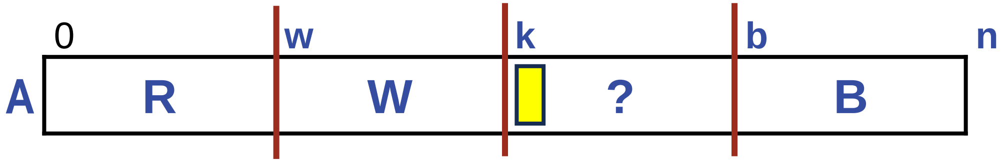


VARIANT: b-k

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.
# -------------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______: _________
```

☞ **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.**



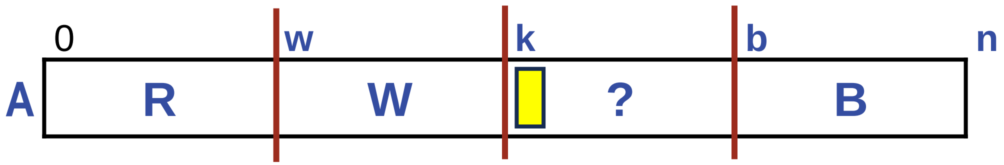
VARIANT: b-k

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.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == ____: __________
    elif 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.**

Dutch National Flag

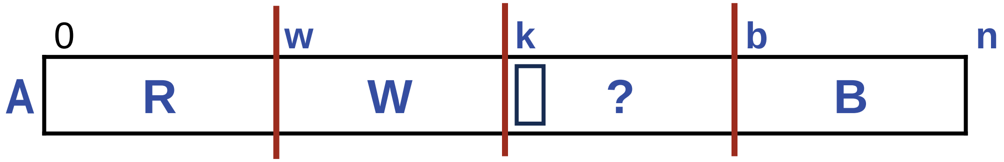


VARIANT: b-k

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.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == B: __________
    elif A[k] == R: __________
    else: __________  # A[k] == W
```

☞ **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.**



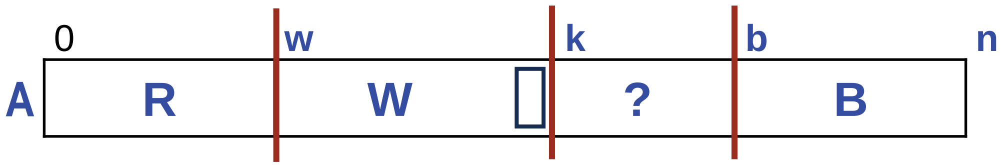
VARIANT: b-k

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.
# --------------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == B: __________
    elif A[k] == R: __________
    else: __________   # A[k] == W
```

☞ **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.**

Dutch National Flag



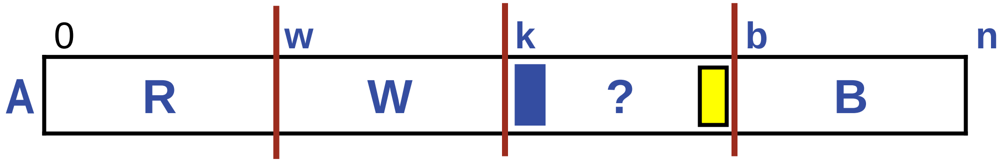
VARIANT: b-k

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.
# -------------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == B: __________
    elif A[k] == R: __________
    else: k += 1  # A[k] == W
```

☞ **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.**

Dutch National Flag

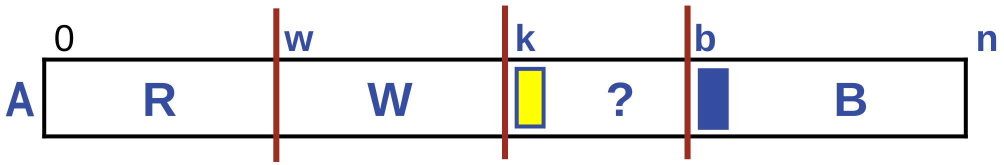


VARIANT: b-k

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.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == B: __________
    elif A[k] == R: __________
    else: k += 1  # A[k] == W
```

☞ **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.**



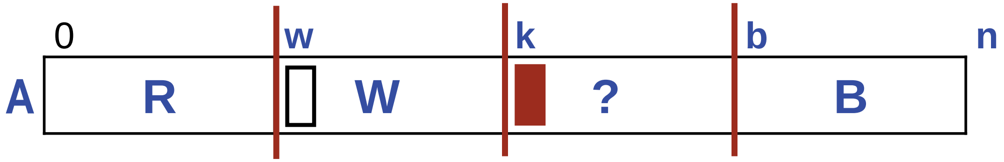
VARIANT: b-k

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.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == B:
        #.Swap A[b-1] and A[k].
        b -= 1
    elif A[k] == ____: __________
    else: k += 1  # A[k] == W
```

☞ **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.**

Dutch National Flag

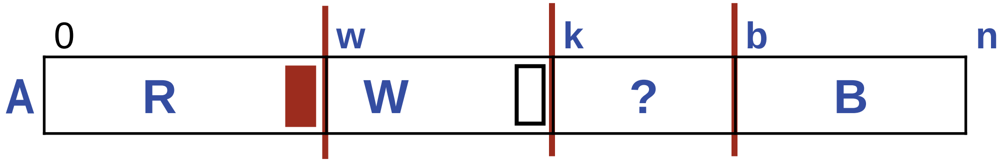


VARIANT: b-k

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.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == B:
        #.Swap A[b-1] and A[k].
        b -= 1
    elif A[k] == R: __________
    else: k += 1  # A[k] == W
```

☞ **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.**



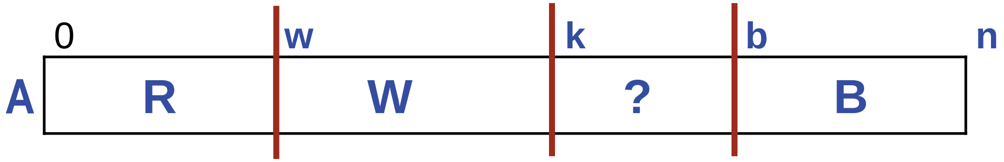
VARIANT: b-k

Dutch National Flag

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.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while ______:
    if A[k] == B:
        #.Swap A[b-1] and A[k].
        b -= 1
    elif A[k] == R:
        #.Swap A[w] and A[k].
        w += 1; k += 1
    else: k += 1  # A[k] == W
```

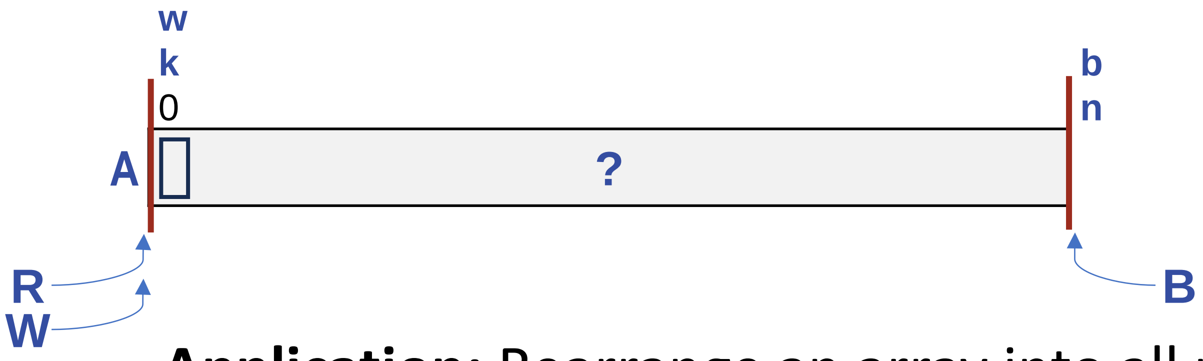
Dutch National Flag



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.
# ----------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = ____;  w = ____   b = ____

while k != b:
    if A[k] == B:
        #.Swap A[b-1] and A[k].
        b -= 1
    elif A[k] == R:
        #.Swap A[w] and A[k].
        w += 1; k += 1
    else: k += 1  # A[k] == W
```



VARIANT: b-k

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

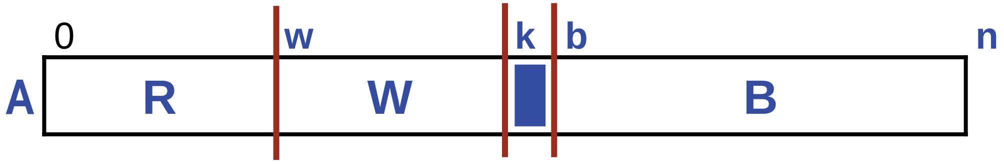
```
# Given array A[0..n-1] consisting of only three values (red, white, and blue),
#   rearrange A into all red, then white, then blue.
# ---------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = 0;  w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

while k != b:
    if A[k] == B:
        #.Swap A[b-1] and A[k].
        b -= 1
    elif A[k] == R:
        #.Swap A[w] and A[k].
        w += 1; k += 1
    else: k += 1  # A[k] == W
```

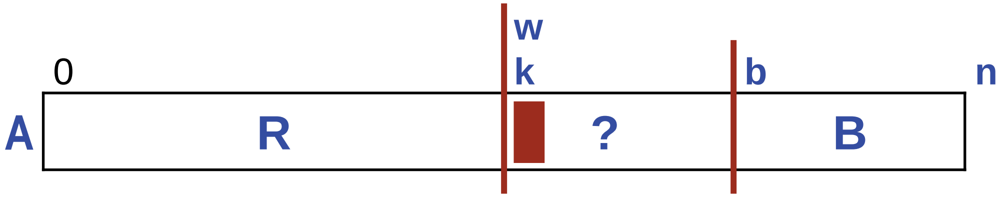
Deferring low-level details has promoted thinking at a higher level of abstraction, but don't forget to elaborate "all the way down".

Dutch National Flag

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.
# ---------------------------------------------------------------------------------
# INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
k = 0;  w = 0;  b = n

while k != b:
    if A[k] == B:
        # Swap A[b-1] and A[k].
        temp = A[b - 1];  A[b - 1] = A[k];  A[k] = temp

        b -= 1
    elif A[k] == R:
        # Swap A[w] and A[k].
        temp = A[k];  A[k] = A[w];  A[w] = temp

        w += 1;  k += 1
    else: k += 1  # A[k] == W
```





Boundary Conditions:

What about potential boundary conditions that might have needed special attention?

You can systematically review the code for issues, but will discover that all are nicely treated by the general case.

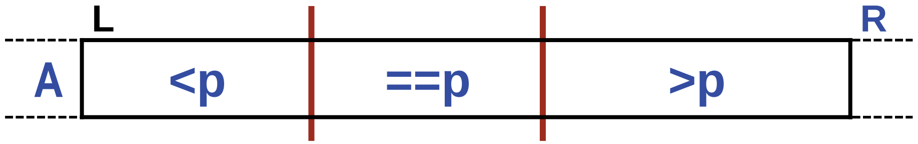
In two situations, `A[k]` gets swapped with itself, but this only happens when its value is already acceptable where it is:

- When the last remaining element in the ? region is blue.
- When `A[k]` is red, and the white region is empty.

All is well.

Performance:

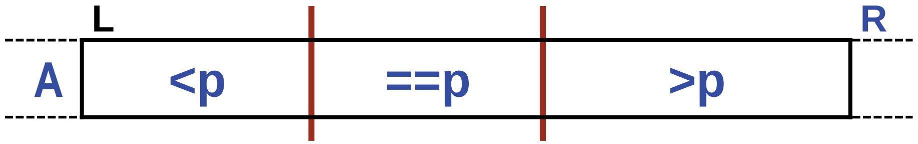
- Constant work per iteration.
- Variant reduced by 1 on each iteration.
- Thus, running time linear in n.



New Application: Rearrange (a segment of) an array into <p, ==p, and >p sections.

```
def partition(A: list[int], L: int, R: int, p: int) -> None:
    """
    Given A[L..R-1] and pivot value p, partition(A,L,R,p) rearranges A[L..R-1] into
    all <p, then all ==p, then all >p.
    """

    ________
```

☞ **A header-comment says exactly what a method must accomplish, not how it does so.**

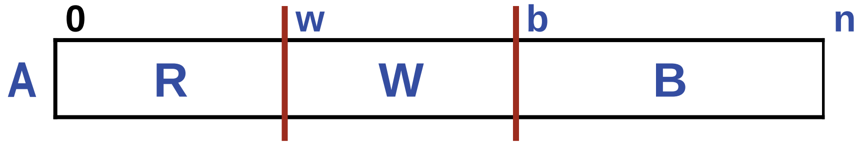


Application: Rearrange (a segment of) an array into <p, ==p, and >p sections.

```
def partition(A: list[int], L: int, R: int, p: int) -> None:
    """
    Given A[L..R-1] and pivot value p, partition(A,L,R,p) rearranges A[L..R-1] into
    all <p, then all ==p, then all >p.
    """

    ⟨body of Dutch National Flag problem⟩
```

We are simply going to replace the *colors* R, W, and B, by the *properties* “<p”, “==p”, and “>p”.

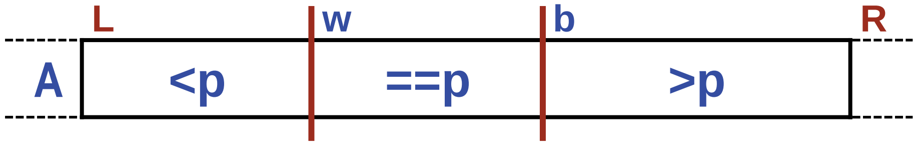
☞ **Don’t type if you can avoid it; clone. Cut and paste, then adapt.**



Application: Rearrange (a segment of) an array into <p, ==p, and >p.

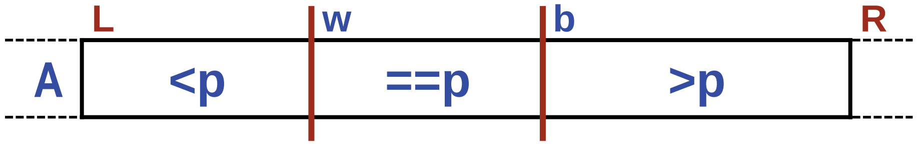
```
def partition(A: list[int], L: int, R: int, p: int) -> None:
    """
    Given A[L..R-1] and pivot value p, partition(A,L,R,p) rearranges A[L..R-1] into
    all <p, then all ==p, then all >p.
    """

    # INVARIANT: A[0..w-1] red, A[w..k-1] white, A[b..n-1] blue, for 0≤w≤k≤b≤n.
    k = 0;  w = 0;  b = n

    while k != b:
        if A[k] == B:
            # Swap A[b-1] and A[k].
            temp = A[b - 1];  A[b - 1] = A[k];  A[k] = temp

            b -= 1
        elif A[k] == R:
            # Swap A[w] and A[k].
            temp = A[k];  A[k] = A[w];  A[w] = temp

            w += 1; k += 1
        else: k += 1  # A[k] == W
```

Dutch National Flag problem



Application: Rearrange (a segment of) an array into <p, ==p, and >p.

```
def partition(A: list[int], L: int, R: int, p: int) -> None:
    """
    Given A[L..R-1] and pivot value p, partition(A,L,R,p) rearranges A[L..R-1] into
    all <p, then all ==p, then all >p.
    """

    # INVARIANT: A[0..w-1] is "<p", A[w..k-1] is "==p", A[b..n-1] is ">p", for 0≤w≤k≤b≤n.
    k = L;  w = L;  b = R

    while k != b:
        if A[k] > p:
            # Swap A[b-1] and A[k].
            temp = A[b - 1];  A[b - 1] = A[k];  A[k] = temp

            b -= 1
        elif A[k] < p:
            # Swap A[w] and A[k].
            temp = A[k];  A[k] = A[w];  A[w] = temp

            w += 1; k += 1
        else: k += 1  # A[k] == p
```



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?

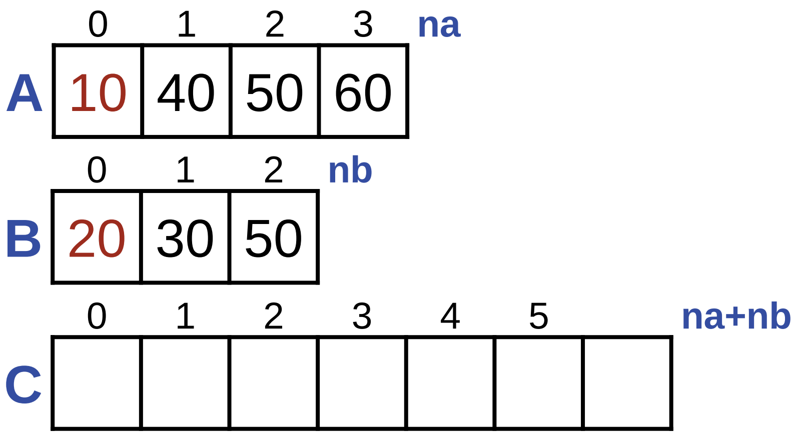
If `A[L..R-1]` are in random order, any of those values is equally good for `p`.

Choosing a p of `(A[L]+A[R-1])/2` guards against poor performance when `A[L..R-1]` is already ordered.

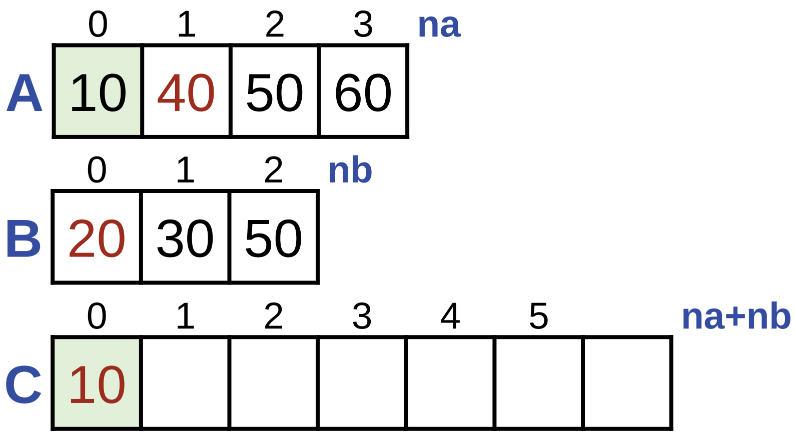
This is central to a Divide and Conquer approach to sorting called QuickSort.

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

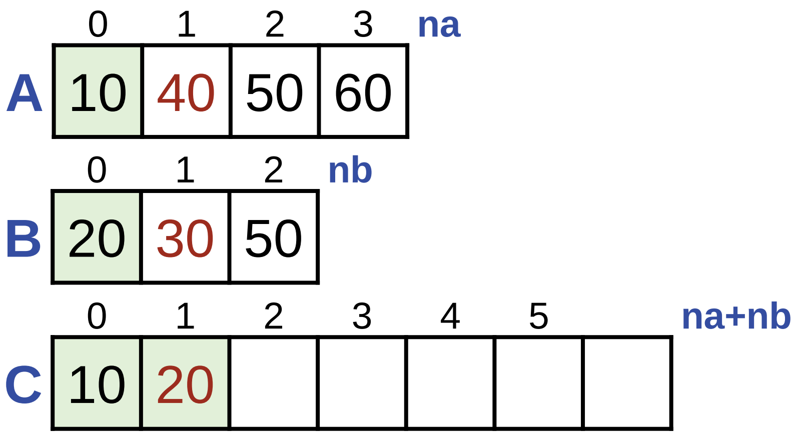
```
#.Given ordered arrays A and B of lengths na and nb, create ordered
#   array C of length na+nb consisting of those values.
```



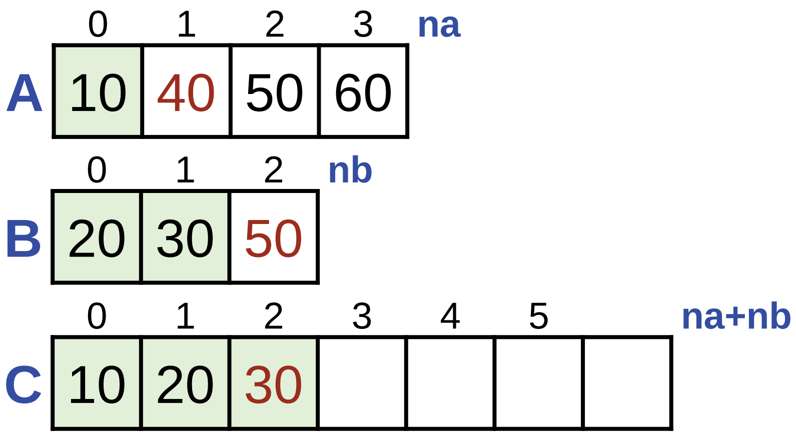
Application: Collate ordered arrays A and B into array C.



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

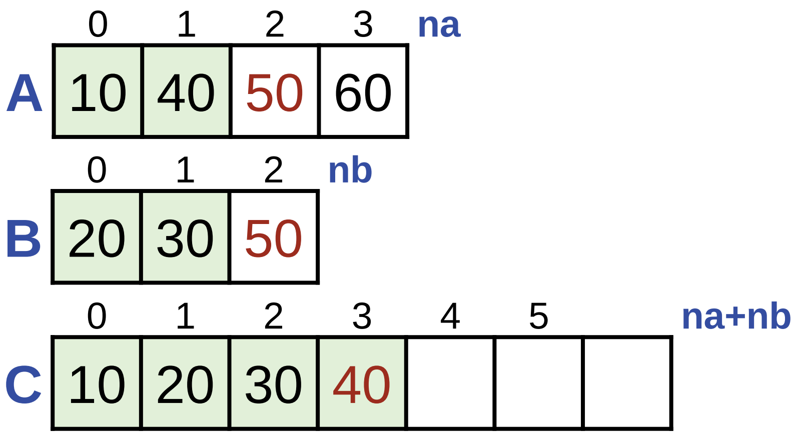


Application: Collate ordered arrays A and B into array C.

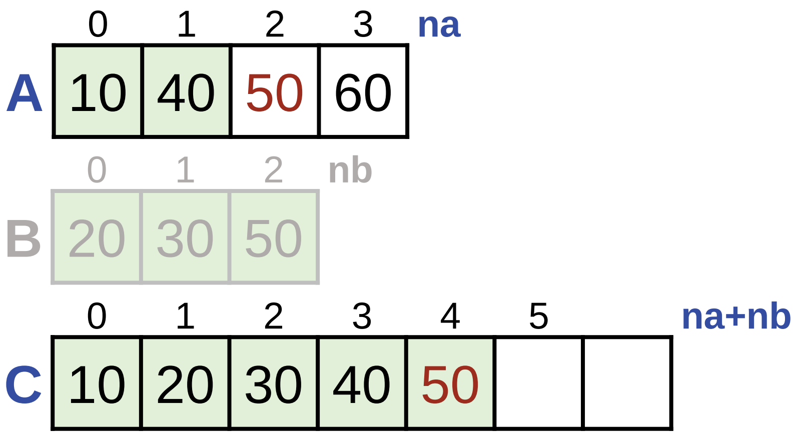


Application: Collate ordered arrays A and B into array C.

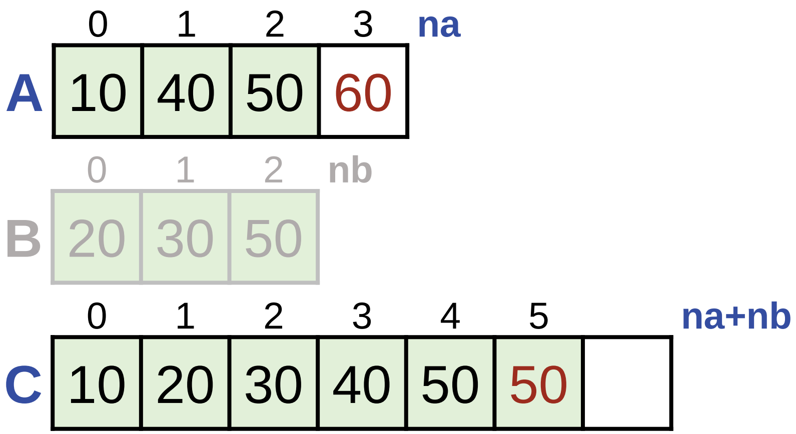
Collation



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

	0	1	2	3	na
A	10	40	50	60	

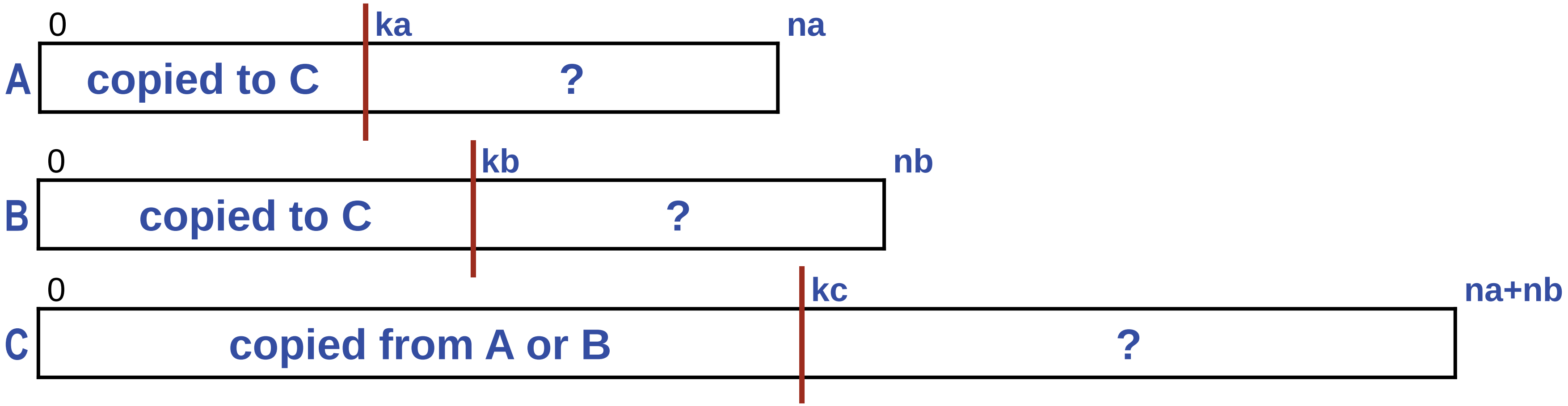
	0	1	2	nb
B	20	30	50	

	0	1	2	3	4	5		na+nb
C	10	20	30	40	50	50	60	

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

Collation is central to a Divide and Conquer approach to sorting called MergeSort.

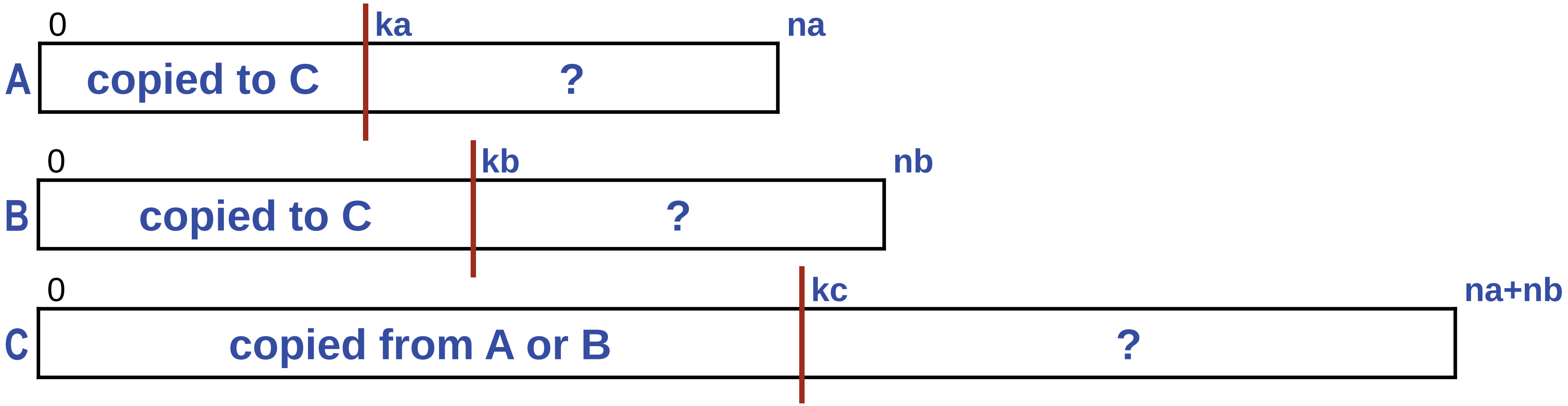
Collation



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.
# -----------------------------------------------------------------
C = [0] * (na + nb)                    # C[0..kc-1] is collation of
                                       #   A[0..ka-1] and B[0..kb-1].
ka = ___;  kb = ___;  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.
```

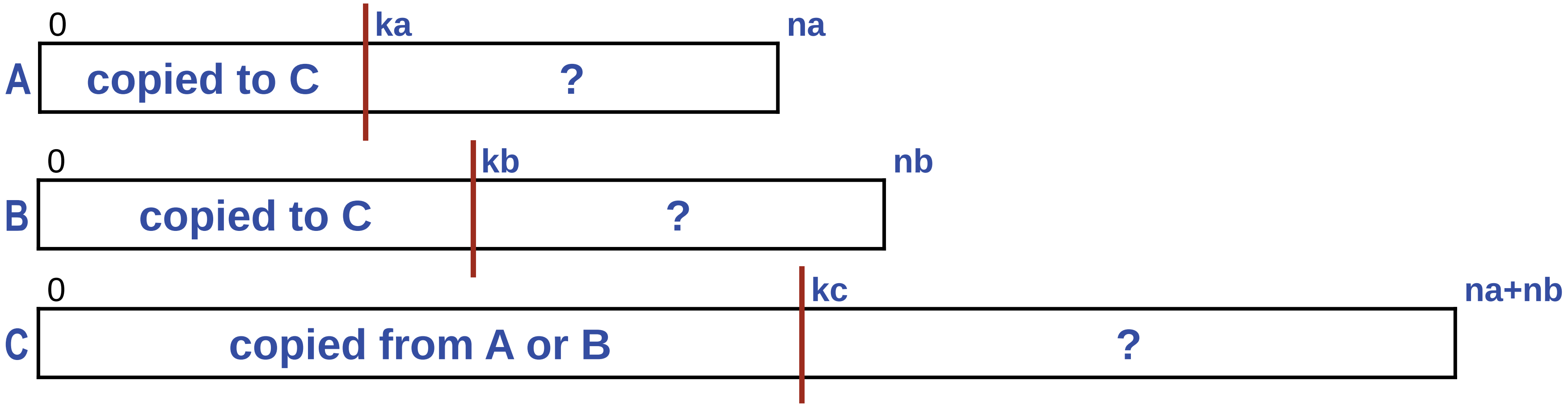
Collation



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

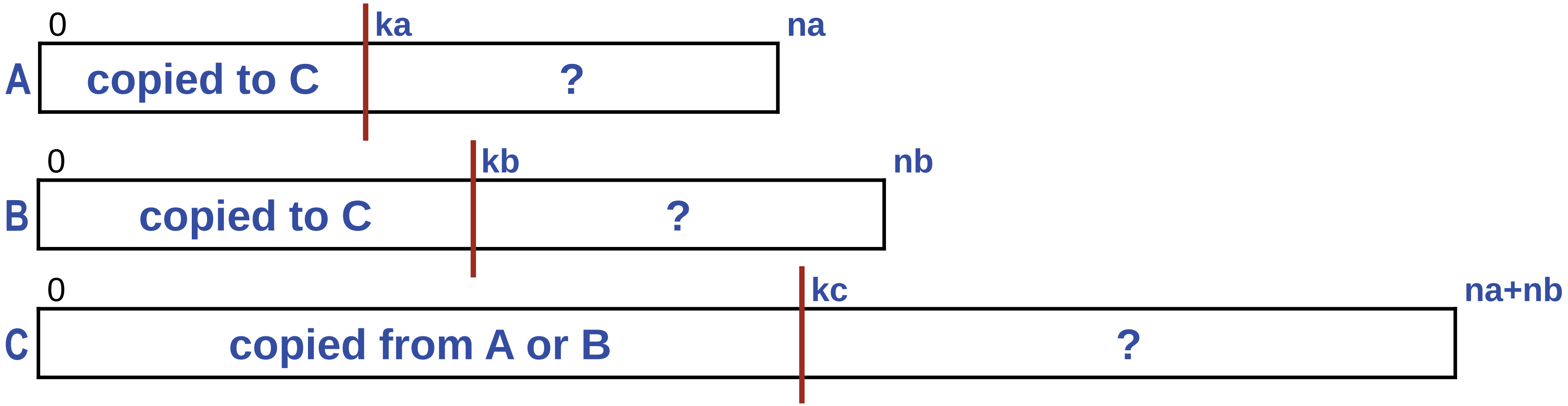
```
# Given ordered arrays A and B of lengths na and nb, create ordered
#   array C of length na+nb consisting of those values.
# -----------------------------------------------------------------
C = [0] * (na + nb)                  # C[0..kc-1] is collation of
                                     #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

Collation



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

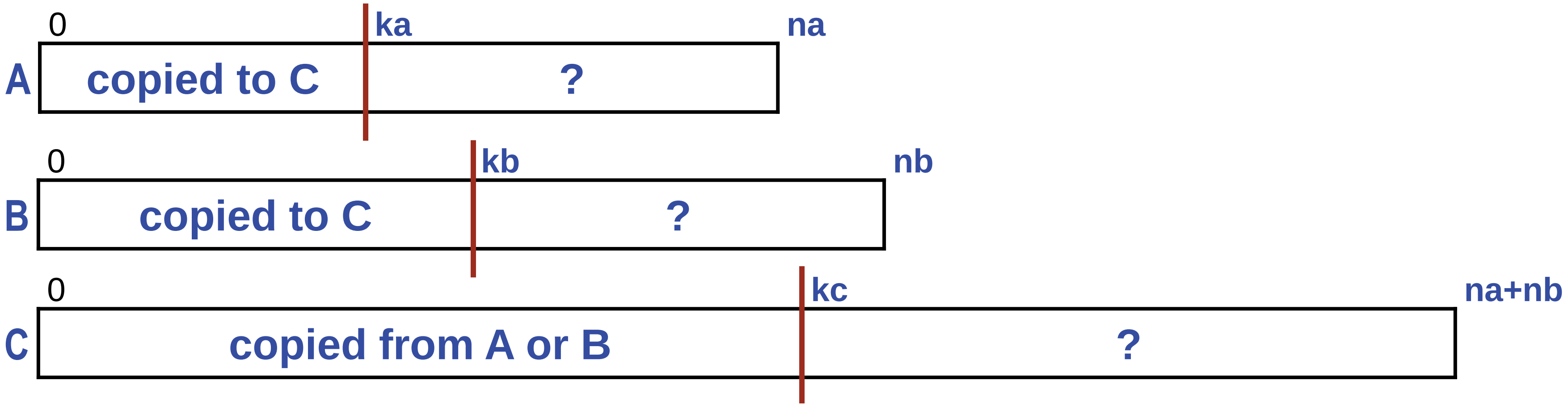
```
# Given ordered arrays A and B of lengths na and nb, create ordered
#   array C of length na+nb consisting of those values.
# ----------------------------------------------------------------
C = [0] * (na + nb)                 # C[0..kc-1] is collation of
                                    #   A[0..ka-1] and B[0..kb-1].
ka = ___;  kb = ___;  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 += 1;  kc += 1
    else:             __________________________

#.Copy remaining values into C from the unexhausted array, A or B.
```



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

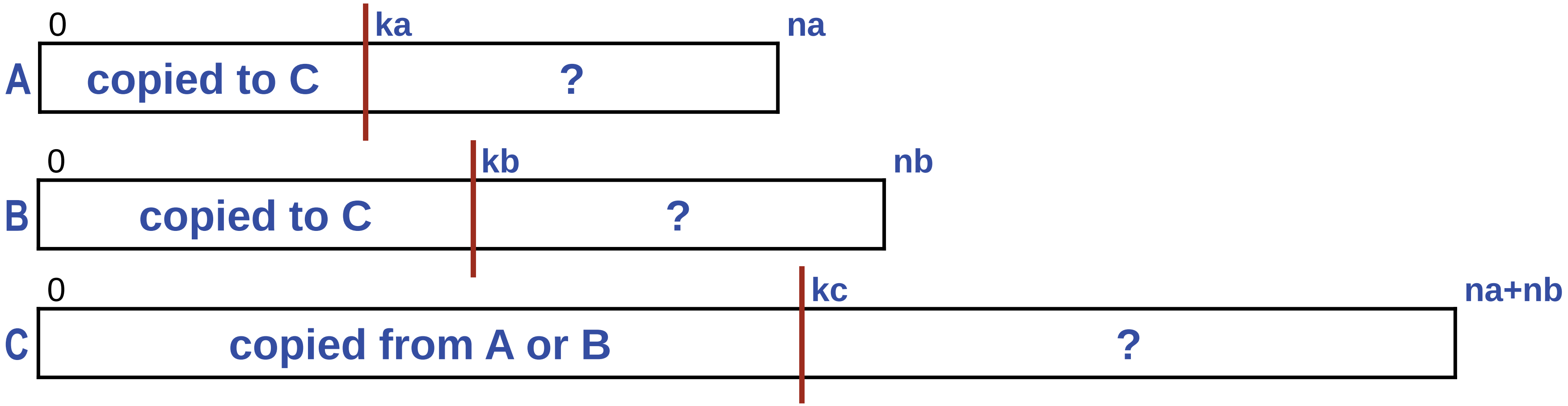
```
# Given ordered arrays A and B of lengths na and nb, create ordered
#   array C of length na+nb consisting of those values.
# ----------------------------------------------------------------
C = [0] * (na + nb)                  # C[0..kc-1] is collation of
                                     #   A[0..ka-1] and B[0..kb-1].
ka = ___;  kb = ___;  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 += 1;  kc += 1
    else:             C[kc] = B[kb];  kb += 1;  kc += 1

#.Copy remaining values into C from the unexhausted array, A or B.
```

Collation



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

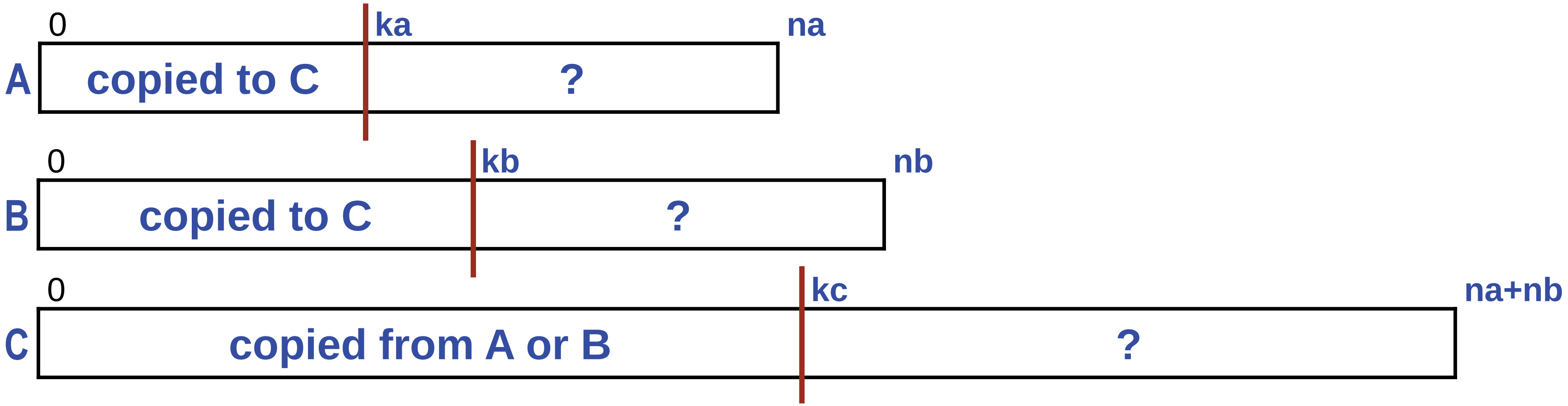
```
# Given ordered arrays A and B of lengths na and nb, create ordered
#   array C of length na+nb consisting of those values.
# -----------------------------------------------------------------
C = [0] * (na + nb)                  # C[0..kc-1] is collation of
                                     #   A[0..ka-1] and B[0..kb-1].
ka = ___;  kb = ___;  kc = ___       # Indices in A, B, and C.

# Copy values from A or B into C until one array is exhausted.
while ______________________:
    if A[ka] < B[kb]: C[kc] = A[ka];  ka += 1;  kc += 1
    else:             C[kc] = B[kb];  kb += 1;  kc += 1

#.Copy remaining values into C from the unexhausted array, A or B.
```



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

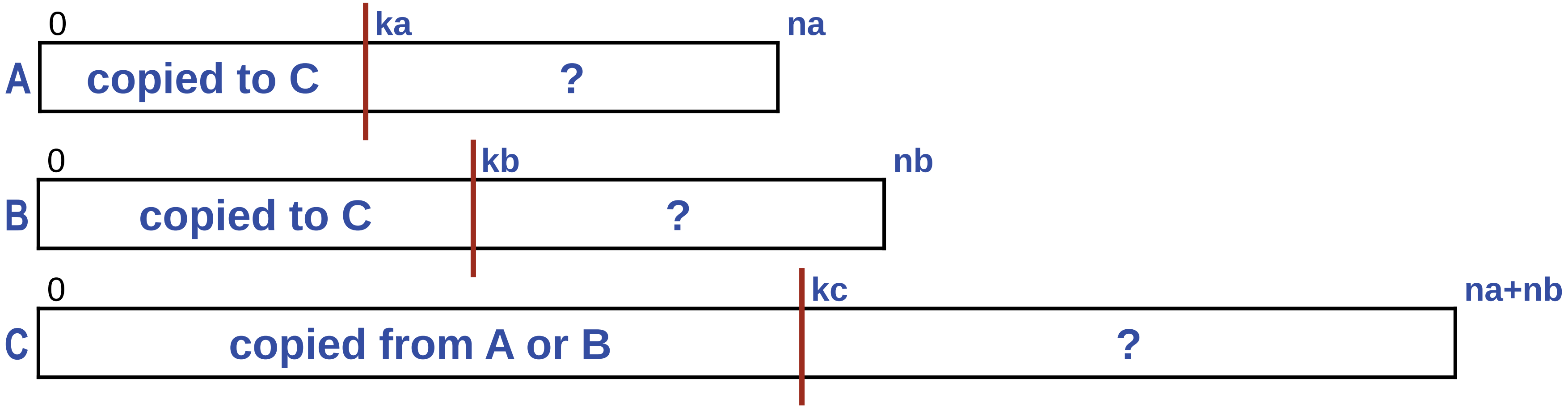
```
# Given ordered arrays A and B of lengths na and nb, create ordered
#   array C of length na+nb consisting of those values.
# -----------------------------------------------------------------
C = [0] * (na + nb)                  # C[0..kc-1] is collation of
                                     #   A[0..ka-1] and B[0..kb-1].
ka = ___;  kb = ___;  kc = ___       # Indices in A, B, and C.

# Copy values from A or B into C until one array is exhausted.
while (ka < na) and (kb < nb):
    if A[ka] < B[kb]: C[kc] = A[ka];  ka += 1;  kc += 1
    else:             C[kc] = B[kb];  kb += 1;  kc += 1

#.Copy remaining values into C from the unexhausted array, A or B.
```

Collation



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.
# -----------------------------------------------------------------
C = [0] * (na + nb)                          # C[0..kc-1] is collation of
                                             #   A[0..ka-1] and B[0..kb-1].
ka = 0;  kb = 0;  kc = 0                     # Indices in A, B, and C.

# Copy values from A or B into C until one array is exhausted.
while (ka < na) and (kb < nb):
    if A[ka] < B[kb]: C[kc] = A[ka];  ka += 1;  kc += 1
    else:             C[kc] = B[kb];  kb += 1;  kc += 1

#.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.
# ----------------------------------------------------------------
C = [0] * (na + nb)                    # C[0..kc-1] is collation of
                                       #   A[0..ka-1] and B[0..kb-1].
ka = 0;  kb = 0;  kc = 0               # Indices in A, B, and C.

# Copy values from A or B into C until one array is exhausted.
while (ka < na) and (kb < nb):
    if A[ka] < B[kb]: C[kc] = A[ka];  ka += 1;  kc += 1
    else:             C[kc] = B[kb];  kb += 1;  kc += 1

# Copy remaining values into C from the unexhausted array, A or B.
while ka < na:        C[kc] = A[ka];  ka += 1;  kc += 1
while kb < nb:        C[kc] = B[kb];  kb += 1;  kc += 1
```

Summary:

A number of useful one-dimensional array rearrangements were presented, some as methods, and some as code fragments:

- `reverse(…)`
- `left_shift_k(…)`
- `left_rotate_one(…)`
- Left-Rotate-k Four separate implementations were developed and assessed.
- Dutch National Flag The basis of Partitioning, and an illustration of invariant-driven programming.
- `partition(…)` The basis for QuickSelect (Chapter 10), and QuickSort (Chapter 11), and an introduction to an algorithm with good *average-case* performance, and not such good *worst-case* performance.
- Collation The basis for MergeSort (Chapter 11).