Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

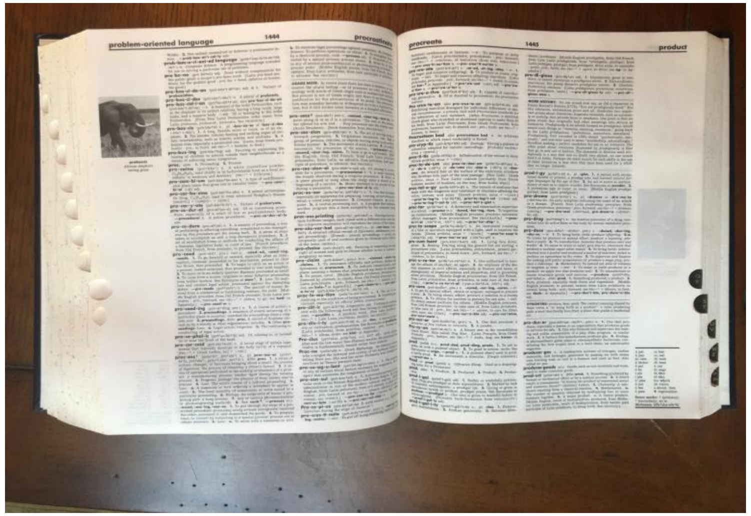
Emeritus Professor
Department of Computer Science
Cornell University

Binary Search

Introduction

If you want to find the definition of the word **proboscis** in a 512-page dictionary, you wouldn't use Sequential Search starting on the first page, say, with **aardvark**. Rather, you would start roughly in the middle. From there, you would:

- Repeatedly halve the portion of the dictionary that remains under consideration, doing so by looking at the middle page of the region in hand, and discarding whichever half is revealed thereby to not contain **proboscis**.
- Once the search has been narrowed to a single page, you would look on that page to see if **proboscis** is there.
- If it is, you found its definition; otherwise, it isn't in the dictionary.



The method is called Binary Search, and is an example of a Divide and Conquer algorithm. Binary Search is astoundingly fast.

Application: Search for a value v in an unordered array `A[0..n-1]`.

```
#.Given array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
```

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

Application: Search for a value `v` in an ordered array `A[0..n-1]`.

```
#.Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A. */
```

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

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
k = 0
while (A[k] != v) and (k < n): k += 1
```

☞ **Master stylized code patterns, and use them.**

Sequential search works, but ignores the order. We can do better.

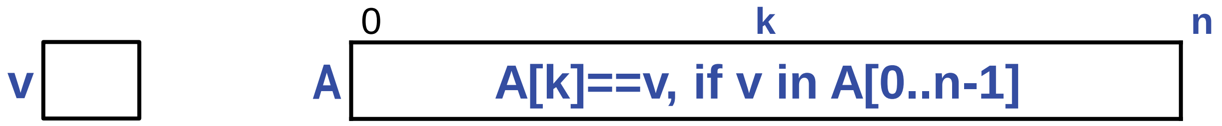
Application: Search for a value v in an ordered array A[0..n-1].

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.

__________
while ______: ___________
__________
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

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



POST

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
__________
while ______: ___________
__________
```

☞ **Invent (or learn) diagrammatic ways to express concepts.**

INVARIANT

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.

__________
while ______: ___________
__________
```

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

Binary Search

INVARIANT

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
while ______: ___________
__________
```

☞ **Introduce program variables whose values describe “state”.**

VARIANT: R-L

INVARIANT

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
while ______:
    if _________: ___________
    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: R-L

INVARIANT

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
while ______:
    if _________:
        R = ______          # Select left “half”.
    else:
        L = ______          # Select right “half”.
__________
```

☞ **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: R-L

INVARIANT

Application: Search for a value v in an ordered array A[0..n-1].

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = ____________       # Compute "midpoint".
while ______:
    if _________:
        R = ______         # Select left "half".
    else:
        L = ______         # Select right "half".
__________
```

☞ **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.**

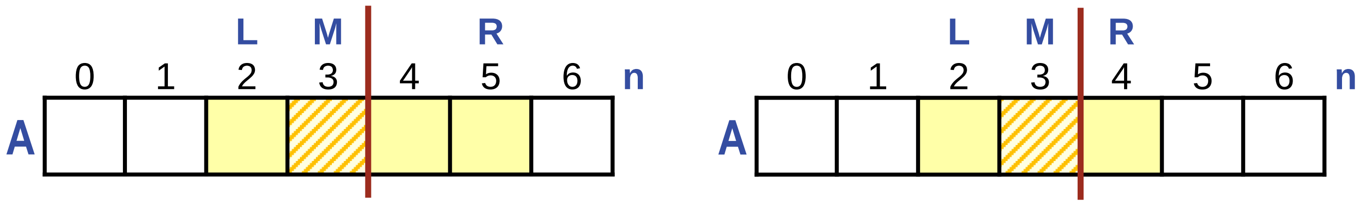
VARIANT: R-L

INVARIANT

Application: Search for a value v in an ordered array A[0..n-1].

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = ____________      # Compute "midpoint".
while ______:
    if _________:
        R = ______        # Select left "half".
    else:
        L = ______        # Select right "half".
__________
```

If you object to A[L..R] straddling the midpoint of A[0..n-1], understand that in "schematic diagrams", the exact locations of boundaries are immaterial.

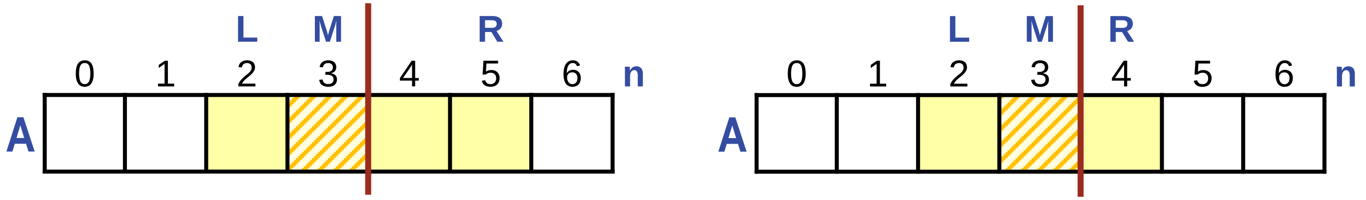


Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = ____________         # Compute "midpoint".
while ______:
    if _________:
        R = ______           # Select left "half".
    else:
        L = ______           # Select right "half".
__________
```

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

Recognize that regions of even and odd lengths may need **distinct** treatments.



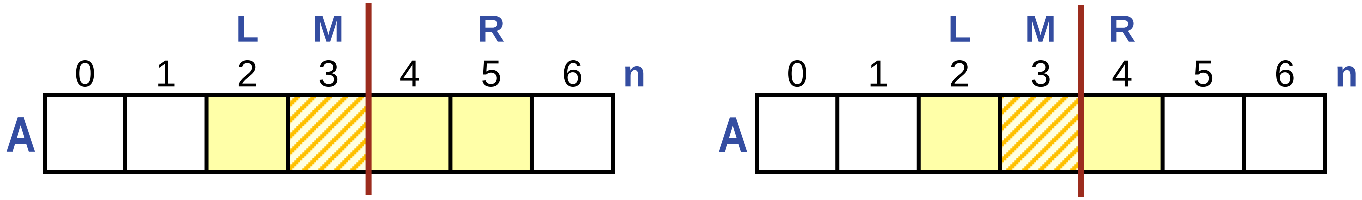
M is index of rightmost element of left "half".

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2       # Compute "midpoint".
while ______:
    if _________:
        R = ______           # Select left "half".
    else:
        L = ______           # Select right "half".
__________
```

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

Recognize that regions of even and odd lengths may need **distinct** treatments, but hope for a **uniform** treatment.

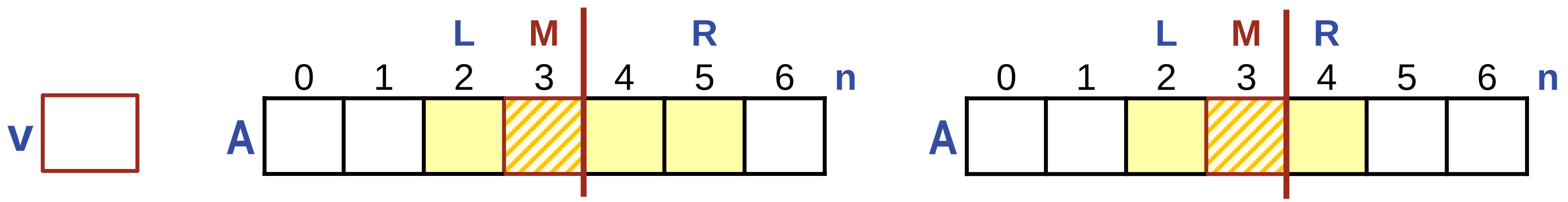


M is index of rightmost element of left "half".

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2      # Compute "midpoint".
while ______:
    if _________:
        R = M              # Select left "half".
    else:
        L = M + 1          # Select right "half".
__________
```

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



M is index of rightmost element of left "half".

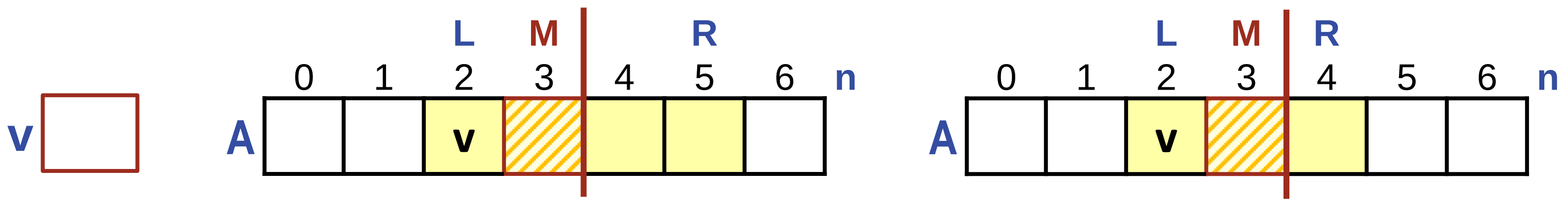
Application: Search for a value v in an ordered array A[0..n-1].

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2       # Compute "midpoint".
while ______:
    if v __ A[M]:
        R = M              # Select left "half".
    else:
        L = M + 1          # Select right "half".
```

____

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

Binary Search



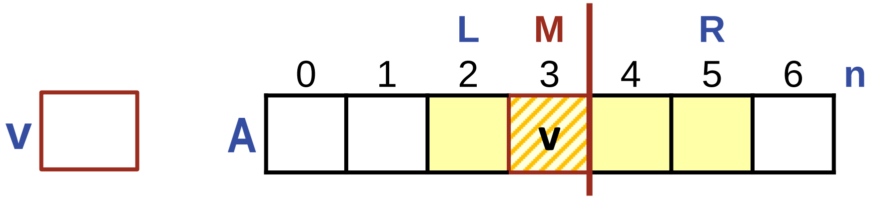
M is index of rightmost element of left “half”.

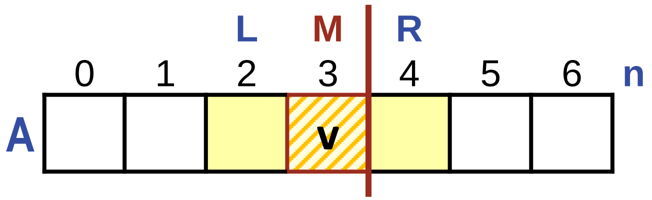
Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2       # Compute “midpoint”.
while ______:
    if v <= A[M]:
        R = M                # Select left “half”.
    else:
        L = M + 1            # Select right “half”.

__________
```

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



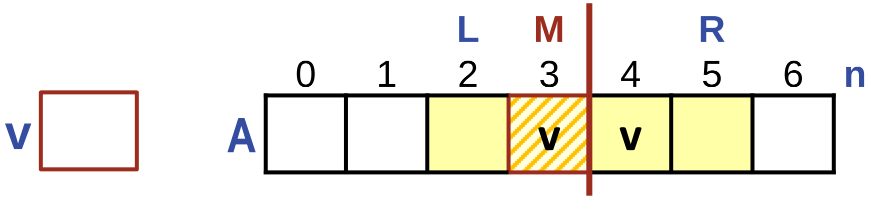


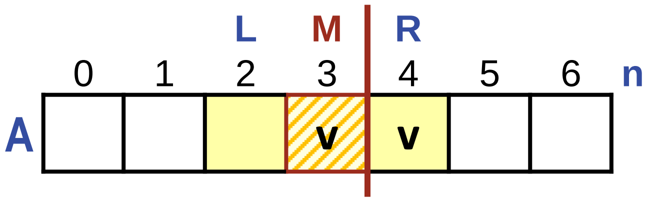
M is index of rightmost element of left "half".

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2      # Compute "midpoint".
while ______:
    if v <= A[M]:
        R = M                # Select left "half".
    else:
        L = M + 1            # Select right "half".
__________
```

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





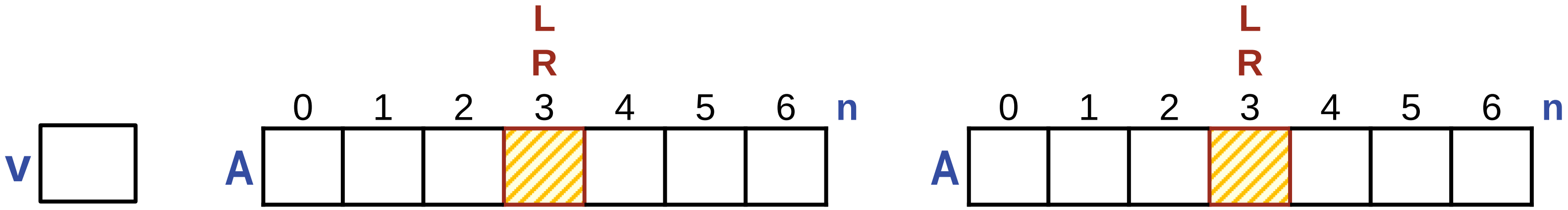
M is index of rightmost element of left “half”.

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2       # Compute “midpoint”.
while ______:
    if v <= A[M]:
        R = M                # Select left “half”.
    else:
        L = M + 1            # Select right “half”.
__________
```

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

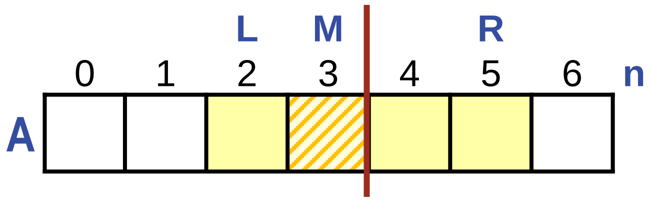
Duplicate instances of v in `A[L..R]` may escape, but not all of them.



Application: Search for a value v in an ordered array A[0..n-1].

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2       # Compute “midpoint”.
while L != R:
    if v <= A[M]:
        R = M              # Select left “half”.
    else:
        L = M + 1          # Select right “half”.
__________
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions



VARIANT:

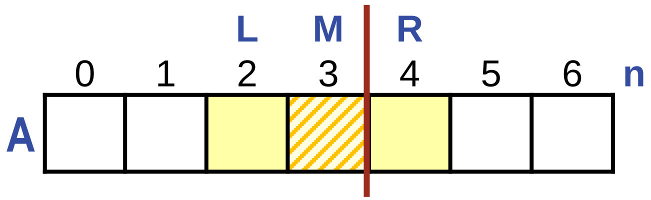
Before			After (left)			After (right)		
L	R	R-L	L	R	R-L	L	R	R-L
2	5	3	2	3	1	4	5	1

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2      # Compute "midpoint".
while L != R:
    if v <= A[M]:
        R = M                # Select left "half".
    else:
        L = M + 1            # Select right "half".
__________
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

Confirm that the **VARIANT** decreases on every iteration.



VARIANT:

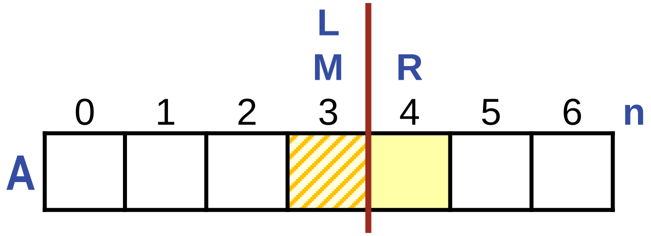
Before			After (left)			After (right)		
L	R	R-L	L	R	R-L	L	R	R-L
2	4	2	2	3	1	4	4	0

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2      # Compute "midpoint".
while L != R:
    if v <= A[M]:
        R = M                # Select left "half".
    else:
        L = M + 1            # Select right "half".
__________
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

Confirm that the **VARIANT** decreases on every iteration.



VARIANT:

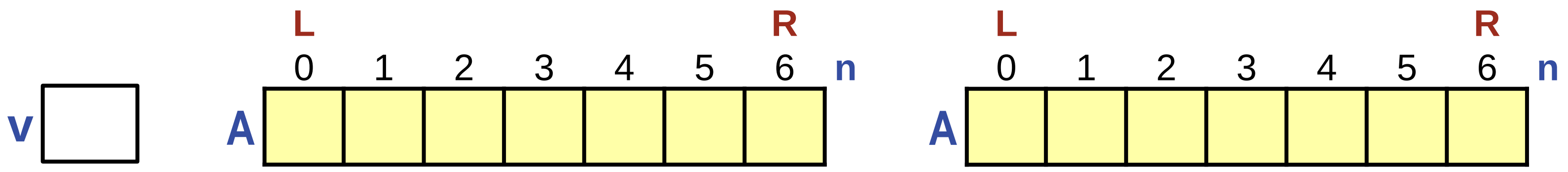
Before			After (left)			After (right)		
L	R	R-L	L	R	R-L	L	R	R-L
3	4	1	3	3	0	4	4	0

Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = ______;  R = ______
M = (L + R) // 2        # Compute "midpoint".
while L != R:
    if v <= A[M]:
        R = M               # Select left "half".
    else:
        L = M + 1           # Select right "half".
__________
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

Confirm that the **VARIANT** decreases on every iteration.

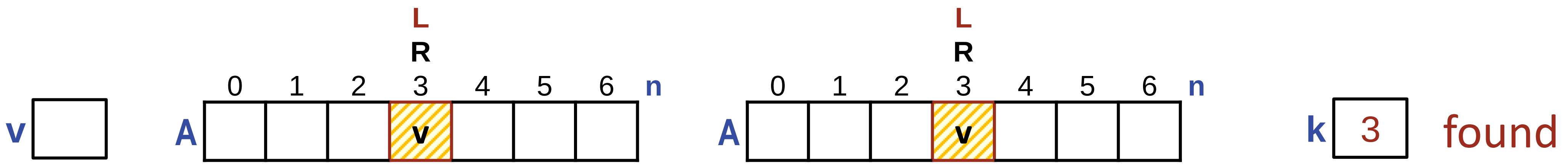


Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = 0;  R = n - 1
M = (L + R) // 2      # Compute “midpoint”.
while L != R:
    if v <= A[M]:
        R = M               # Select left “half”.
    else:
        L = M + 1           # Select right “half”.
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

Binary Search

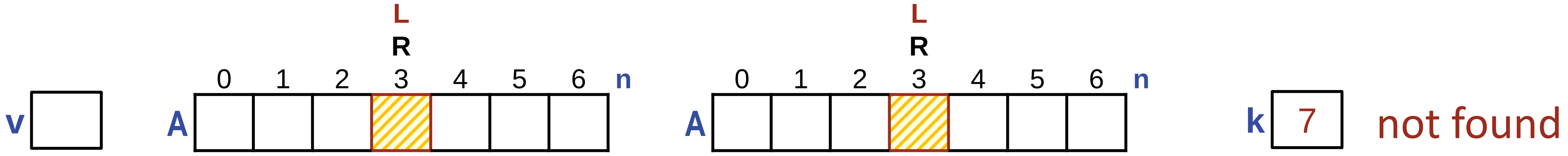


Application: Search for a value v in an ordered array A[0..n-1].

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = 0;  R = n - 1
M = (L + R) // 2      # Compute "midpoint".
while L != R:
    if v <= A[M]:
        R = M               # Select left "half".
    else:
        L = M + 1           # Select right "half".
if A[L] == v: k = L
else: k = n
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

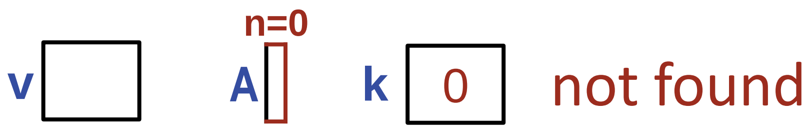
Binary Search



Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
L = 0;  R = n - 1
M = (L + R) // 2      # Compute “midpoint”.
while L != R:
    if v <= A[M]:
        R = M               # Select left “half”.
    else:
        L = M + 1           # Select right “half”.
if A[L] == v: k = L
else: k = n
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

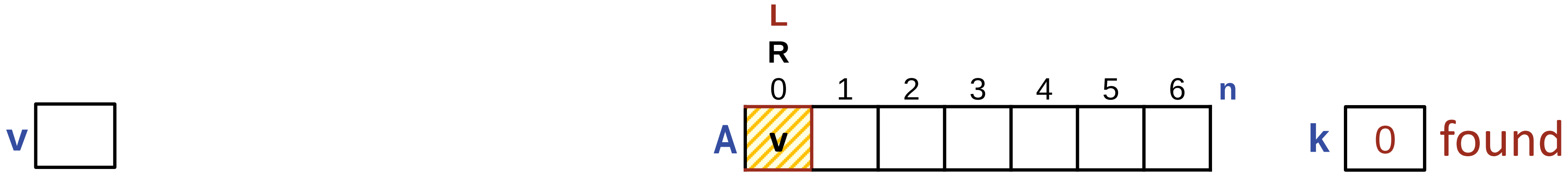


Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
if n == 0: k = 0
else:
    L = 0;  R = n - 1
    M = (L + R) // 2       # Compute “midpoint”.
    while L != R:
        if v <= A[M]:
            R = M                # Select left “half”.
        else:
            L = M + 1            # Select right “half”.
    if A[L] == v: k = L
    else: k = n
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

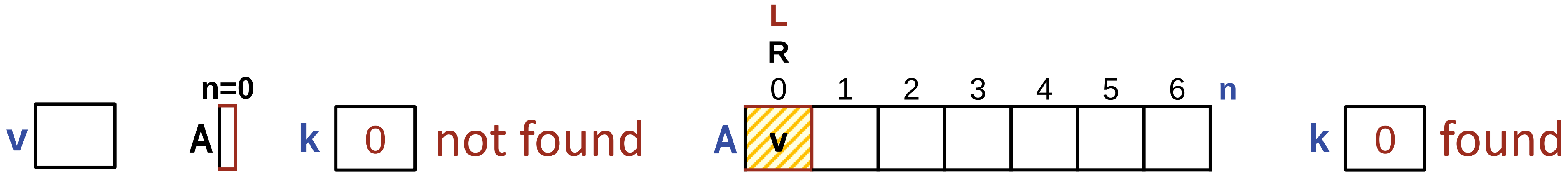
Binary Search



Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
if n == 0: k = 0
else:
    L = 0;  R = n - 1
    M = (L + R) // 2      # Compute “midpoint”.
    while L != R:
        if v <= A[M]:
            R = M                # Select left “half”.
        else:
            L = M + 1            # Select right “half”.
    if A[L] == v: k = L
    else: k = n
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

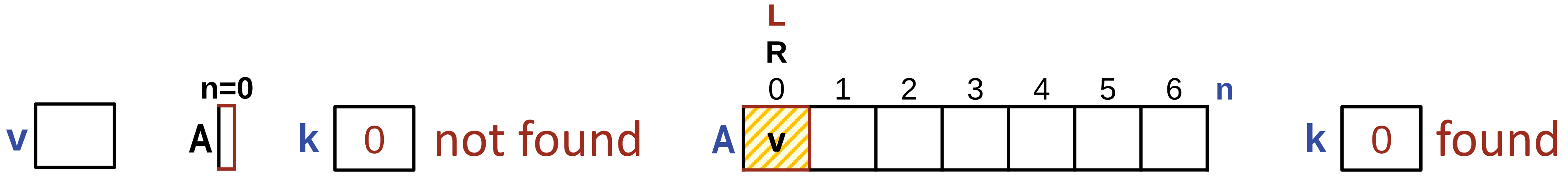


Application: Search for a value v in an ordered array `A[0..n-1]`.

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
if n == 0: k = 0
else:
    L = 0;  R = n - 1
    M = (L + R) // 2      # Compute “midpoint”.
    while L != R:
        if v <= A[M]:
            R = M               # Select left “half”.
        else:
            L = M + 1           # Select right “half”.
    if A[L] == v: k = L
    else: k = n
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

Is it a problem that k==0 represents both “not found” and “found in 0^{th} element”?



Application: Search for a value v in an ordered array A[0..n-1].

```
# Given ordered array A[0..n-1], n≥0, and value v, let k be an index of A
#   where A[k]==v, or n if there is no v in A.
if n == 0: k = 0
else:
    L = 0;  R = n - 1
    M = (L + R) // 2      # Compute “midpoint”.
    while L != R:
        if v <= A[M]:
            R = M             # Select left “half”.
        else:
            L = M + 1         # Select right “half”.
    if A[L] == v: k = L
    else: k = n
```

Coding order
(1) body
(2) termination
(3) initialization
(4) finalization
(5) boundary conditions

```
# Test for found.
    if k<n: # Found.
    else:   # Not found.
```

No. What matters is whether k<n, not whether k==0.

Binary Search is astoundingly fast. If n==512, just 9 iterations to termination!

Iteration #	VARIANT
0	512
1	256
2	128
3	64
4	32
5	16
6	8
7	4
8	2
9	1

Running time is logarithmic in n,

and independent of whether v is in A or not.