Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

Sequential Search

To *search* is to look for something systematically on behalf of a *client*.

The *search-use pattern* is a specialization of the compute-use pattern.

```
/* Search. */
/* Use the search result. */
```

```
/* Compute. */
/* Use. */
```

To *search* is to look for something systematically on behalf of a *client*.

The *search-use pattern* is a specialization of the compute-use pattern.

```
/* Search. */
/* Use the search result. */
```

We *search* for something in a collection of items.

The collection can be unbounded, e.g., natural numbers, or values in a file.
The collection can be bounded, e.g., characters in text, or elements of an array.

Search in an unbounded collection can succeed or run forever, and in a bounded collection can succeed or fail.

Indeterminate-iteration, the mother of all searches, seeks the smallest k≥0 with some property, i.e., negation of the *condition*:

```
/* Search. */
   int k = 0;
   while ( condition ) k++;
```

It is called a sequential search because it checks values one at time, in order.

Indeterminate-iteration, the mother of all searches, seeks the smallest k≥0 with some property, i.e., negation of the *condition*:

```
/* Search. */
   int k = 0;
   while ( condition ) k++;

/* Use k. */
```

It is called a sequential search because it checks values one at time, in order. When it stops, k is the value sought.

Sequential search can be unbounded, or it can be bounded:

```
/* Search. */
int k = 0;
while ( k<=maximum && condition ) k++;

# Use.
if (k <= maximum) /* Found. */
else /* Not Found. */
```

Generalizing, sequential search in a collection sets p to what you are looking for (or where it is), or an indication that it was not found:

```
p = the-first-place-look;
while ( p is-not-beyond-the-last-place-to-look &&
        p is-not-what-you-are-looking-for )
   p = the-next-place-to-look;
if ( p is-not-beyond-the-last-place-to-look ) /* Found. */
else /* Not found. */
```

We consider four applications of sequential search in a collection:

- Primality Testing
- Search in an Unordered Array
- Array Equality
- Longest Descending Suffix

and Find Minimal in an Unordered Array, which isn’t really a sequential search, and contrasts with it.

We consider three applications of sequential search in a collection:

- Primality Testing
- Search in an Unordered Array
- Array Equality
- Longest Descending Suffix

and Find Minimal in an Unordered Array, which isn't really a sequential search, and contrasts with it.

N.B. We have used the term collection loosely. We shall later use the term collection in a more technical sense.

Definition: Natural number p is prime if its only divisors are 1 and p; it is composite otherwise.

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

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

---

2 3 4 5 6 7 8 9 10 11 12 13 14 15 prime

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

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

2 3 4 5 6 7 8 9 10 11 12 13 14 (15) composite

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

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

```
2   3   4   5   6  (7)  8   9 10 11 12 13 14 15
```

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

---

2 3 4 5 6 7 8 9 10 11 12 13 14 15

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

2 3 4 5 6 7 8 9 10 11 12 13 14 15

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

---

2 3 4 5 6 7 8 9 10 11 12 13 14 15

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

2 3 4 5 6 (7) 8 9 10 11 12 13 14 15

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

2 3 4 5 6 (7) 8 9 10 11 12 13 14 15 prime

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

2 3 4 5 6 7 8 9 10 11 12 13 14 15

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

2 3 4 5 6 7 8 9 10 11 12 13 14 (15) composite

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

2 3 4 5 6 7 8 9 10 11 12 13 14 (15) composite

👆

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Searching for the smallest divisor of p that is greater or equal to 2.

2 3 4 5 6 7 8 9 10 11 12 13 14 (15) composite

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. */
   /* Use. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Searching for the smallest divisor of p that is greater or equal to 2.

2 3 4 5 6 7 8 9 10 11 12 13 14 (15) composite

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
   /* Use d. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Searching for the smallest divisor of p that is greater or equal to 2.

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
   if ( ____ ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

☞ **Refine specifications and placeholders in an order that makes sense for development, without regard to execution order.**

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
   if ( d__p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

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

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
   if ( d==p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

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

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
      int d = 2;
      while ( ________ ) d++;
   if ( d==p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

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

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
      int d = 2;
      while ( (p%d)__0 ) d++;
   if ( d==p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

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

Application: Write a program segment to say whether p is prime or composite.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
      int d = 2;
      while ( (p%d)!=0 ) d++;
   if ( d==p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

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

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

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

Search in an Unordered Array

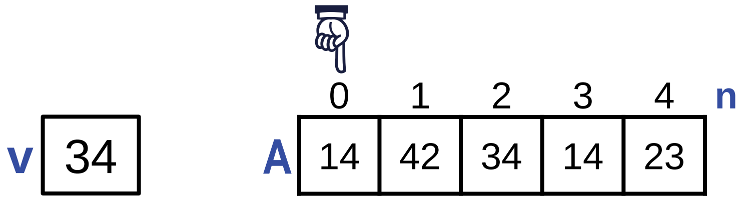
v 34

	0	1	2	3	4	n
A	14	42	34	14	23	

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

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

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

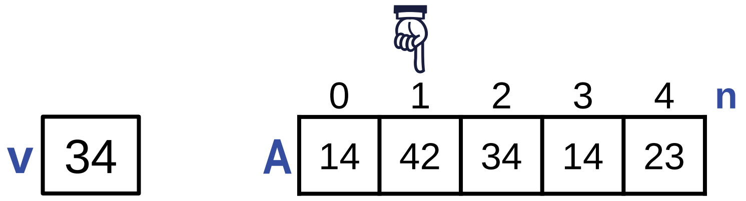


Application: Search for a value v in an unordered array `A[0..n-1]`.

```
/* Find v in A[0..n-1], or indicate it’s not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

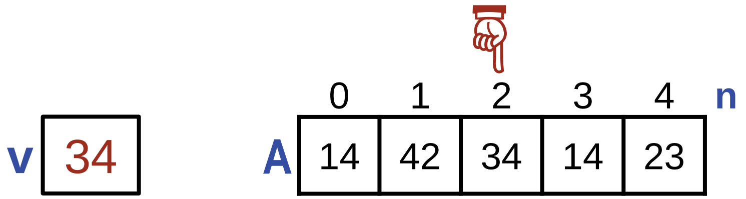
Search in an Unordered Array



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

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**



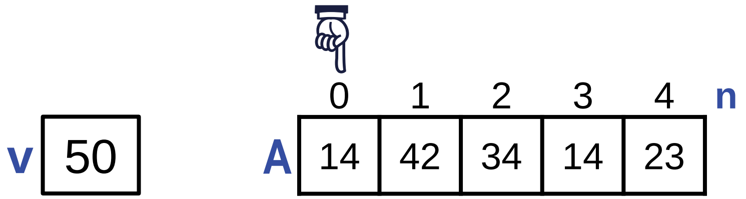
found

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

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Search in an Unordered Array

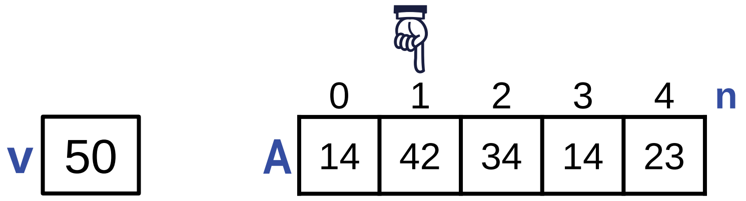


Application: Search for a value v in an unordered array `A[0..n-1]`.

```
/* Find v in A[0..n-1], or indicate it’s not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Search in an Unordered Array

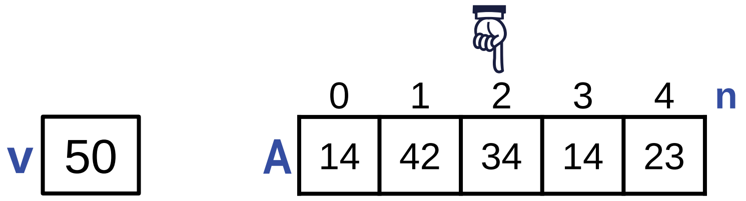


Application: Search for a value v in an unordered array `A[0..n-1]`.

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Search in an Unordered Array

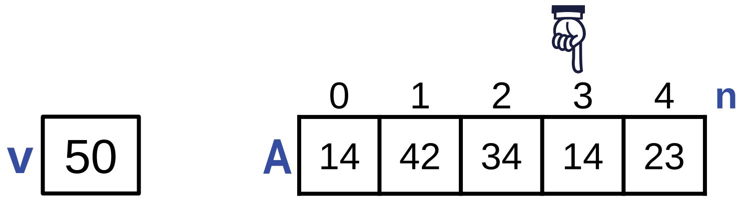


Application: Search for a value v in an unordered array `A[0..n-1]`.

```
/* Find v in A[0..n-1], or indicate it’s not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Search in an Unordered Array

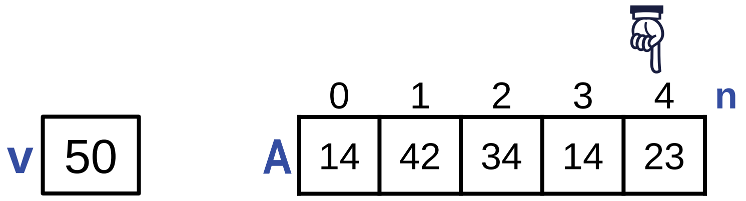


Application: Search for a value v in an unordered array `A[0..n-1]`.

```
/* Find v in A[0..n-1], or indicate it’s not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

Search in an Unordered Array



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

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

v: 50

A	0	1	2	3	4	n
	14	42	34	14	23	

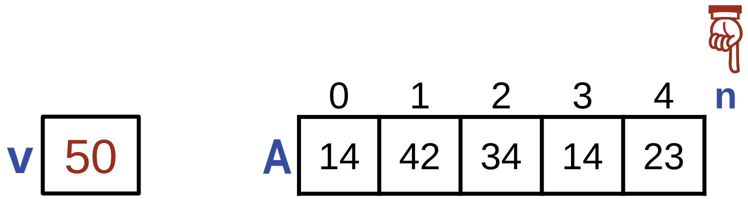
not found

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

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Search in an Unordered Array



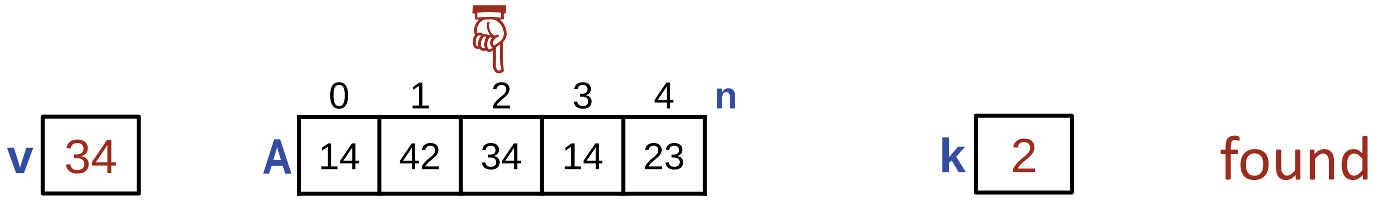
not found

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

```
/* Find v in A[0..n-1], or indicate it's not there. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

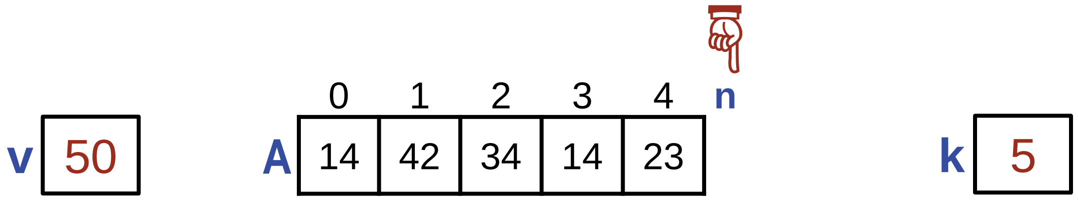
Sequential Search.



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 the smallest non-negative
   integer s.t. A[k]==v. */
```

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



not found

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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
```

☞ **Choose data representations that are uniform, if possible.**

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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( k<=maximum && condition ) k++;
```

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

Search in an Unordered Array

v: 34

A:

	0	1	2	3	4	n
A	14	42	34	14	23	

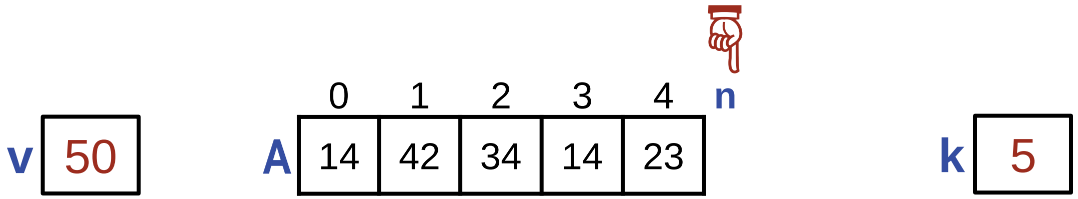
k: 2

found

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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( k<=maximum && A[k]!=v ) k++;
```

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



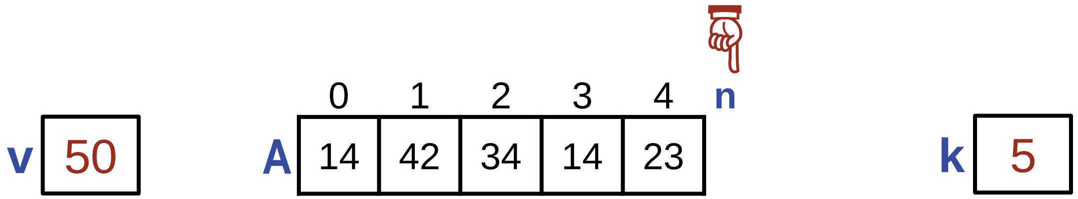
not found

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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( k<=n-1 && A[k]!=v ) k++;
```

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

Search in an Unordered Array

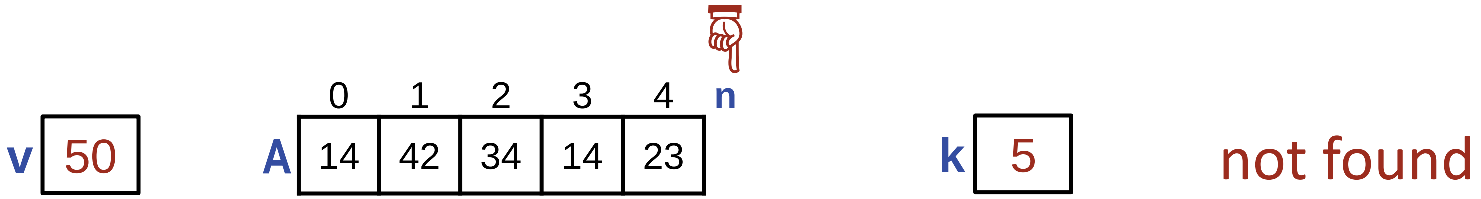


not found

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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( k<n && A[k]!=v ) k++;
```

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



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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( k<n && A[k]!=v ) k++;
```

Short-circuit mode **and**. If left operand is **false**, the right operand is not evaluated, which prevents a “subscript out-of-bounds error”.

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

Search in an Unordered Array

v 50

A

0	1	2	3	4	n
14	42	34	14	23	

k 5

not found

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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( A[k]!=v && k<n ) k++;
```

Short-circuit mode **and**. The reverse order would be incorrect because the "subscript out-of-bounds error" would occur before discovering that k<n is **false**.

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

Search in an Unordered Array

INVARIANT

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 the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( k<n && A[k]!=v ) k++;
```

☞ **Alternate between using a concrete example to guide you in characterizing "program state", and an abstract version that refers to all possible examples.**

New Application: Are arrays `A[0..n-1]` and `B[0..n-1]` equal?

```
/* Given arrays A[0..n-1] and B[0..n-1], set e to true if A equals B,
   else set e to false. */
```

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

Array Equality

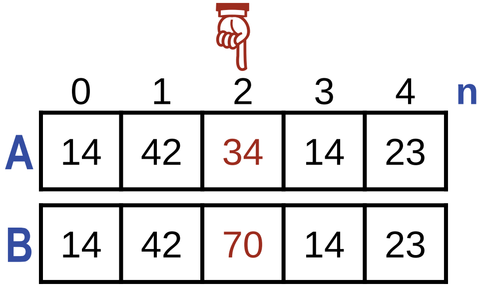
	0	1	2	3	4	n
A	14	42	34	14	23	
B	14	42	34	14	23	

equal

Application: Are arrays `A[0..n-1]` and `B[0..n-1]` equal?

```
/* Given arrays A[0..n-1] and B[0..n-1], set e to true if A equals B,
   else set e to false. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**



not equal

Application: Are arrays `A[0..n-1]` and `B[0..n-1]` equal?

```
/* Given arrays A[0..n-1] and B[0..n-1], set e to true if A equals B,
   else set e to false. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

	0	1	2	3	4	n
A	14	42	34	14	23	
B	14	42	70	14	23	

not equal

Application: Are arrays `A[0..n-1]` and `B[0..n-1]` equal?

```
/* Given arrays A[0..n-1] and B[0..n-1], set e to true if A equals B,
   else set e to false. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Sequential Search for **not** equal.

	0	1	2	3	4	n
A	14	42	34	14	23	
B	14	42	70	14	23	

not equal

Application: Are arrays `A[0..n-1]` and `B[0..n-1]` equal?

```
/* Given arrays A[0..n-1] and B[0..n-1], set e to true if A equals B,
   else set e to false. */
   int k = 0;
   while ( k<=maximum && condition ) k++;
   if ( k<=maximum ) /* Found. */
   else /* Not found. */
```

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

Sequential Search for **not** equal.

	0	1	2	3	4	n
A	14	42	34	14	23	
B	14	42	74	14	23	

not equal

Application: Are arrays A[0..n-1] and B[0..n-1] equal?

```
/* Given arrays A[0..n-1] and B[0..n-1], set e to true if A equals B,
   else set e to false. */
   int k = 0;
   while ( k<=maximum && A[k]==B[k] ) k++;
   if ( k<n ) e = false;
   else /* Not found. */
```

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

Sequential Search for **not** equal.

	0	1	2	3	4	n
A	14	42	34	14	23	
B	14	42	34	14	23	

equal

Application: Are arrays A[0..n-1] and B[0..n-1] equal?

```
/* Given arrays A[0..n-1] and B[0..n-1], set e to true if A equals B,
   else set e to false. */
   int k = 0;
   while ( k<n && A[k]==B[k] ) k++;
   if ( k<n ) e = false;
   else e = true;
```

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

Sequential Search for **not** equal.

Technique: Sentinel search.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
      int d = 2;
      while ( (p%d)!=0 ) d++;
   if ( d==p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

Recall the search for the smallest divisor of p in Primality Testing.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 prime

Technique: Sentinel search.

Q. Why was there no bound check?

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
      int d = 2;
      while ( (p%d)!=0 ) d++;
   if ( d==p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

Recall the search for the smallest divisor of p in Primality Testing.

Sentinel Search

2 3 4 5 6 7 8 9 10 11 12 13 14 15 prime

Technique: Sentinel search.

Q. Why was there no bound check?

A. Because every number is divisible by itself.

```
/* Given p≥2, output whether p is prime or composite. */
   /* Search. Let d≥2 be the smallest divisor of p. */
      int d = 2;
      while ( (p%d)!=0 ) d++;
   if ( d==p ) System.out.println( "prime" );
   else System.out.println( "composite" );
```

Divisibility of every number by itself “stands guard” to prevent going too far.

Sentinel Search

v: 50

	0	1	2	3	4	n
A	14	42	34	14	23	

k: 5

not found

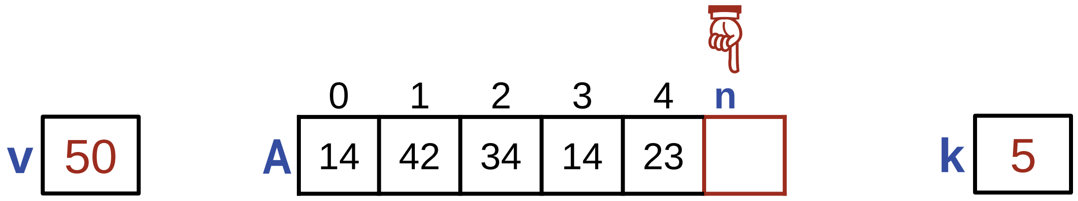
Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int k = 0;
   while ( k<n && A[k]!=v ) k++;
```

Q. How can we obviate this bound check?

Now recall the sequential search for an instance of v in an array A.

Sentinel Search



not found

Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A.
   Assume A[n] exists. */
   int k = 0;
   while ( k<n && A[k]!=v ) k++;
```

Q. How can we obviate this bound check?

Sentinel Search

v: 50

A:

0	1	2	3	4	n
14	42	34	14	23	50

k: 5

not found

Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A.
   Assume A[n] exists. */
   A[n] = v;                              // Stand guard to keep k≤n.
   int k = 0;
   while ( k<n && A[k]!=v ) k++;
```

Q. How can we obviate this bound check?
A. Copy v into A[n].

Sentinel Search

	0	1	2	3	4	n
A	14	42	34	14	23	50

v: 50

k: 5

not found

Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A.
   Assume A[n] exists. */
   A[n] = v;                              // Stand guard to keep k≤n.
   int k = 0;
   while ( A[k]!=v ) k++;
```

Q. How can we obviate this bound check?

A. Copy v into A[n]. Eliminate the check.

v 50

A	0	1	2	3	4	n
	14	42	34	14	23	50

k 5

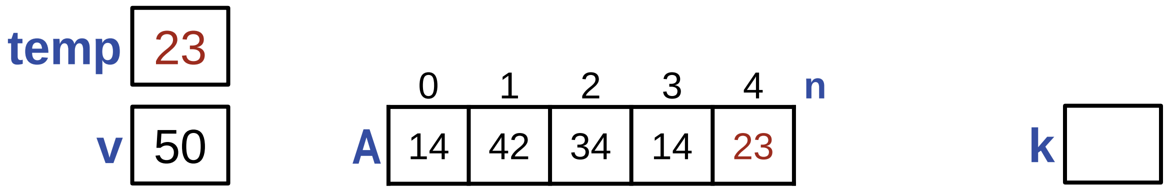
not found

Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A.
   Assume A[n] exists. */
   A[n] = v;                              // Stand guard to keep k≤n.
   int k = 0;
   while ( A[k]!=v ) k++;
```

If you prefer to not assume that A[n] exists,

Sentinel Search

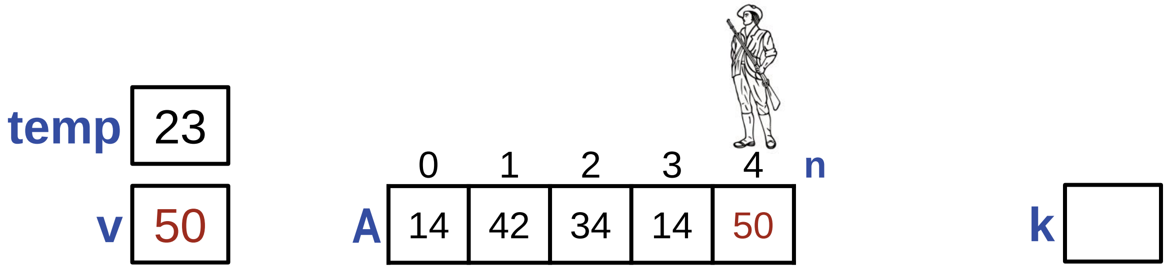


Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int temp = A[n-1];                   // Save A[n-1].
   A[___] = v;                          // Stand guard to keep ___.
   int k = 0;
   while ( A[k]!=v ) k++;
```

If you prefer to not assume that A[n] exists, use A[n-1] for the sentinel, instead. First, save A[n-1] in a temporary variable.

Sentinel Search

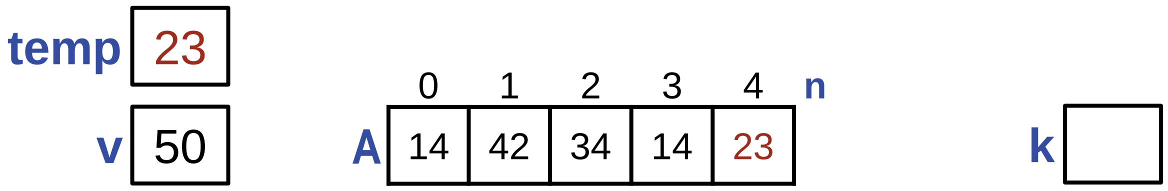


Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int temp = A[n-1];                     // Save A[n-1].
   A[n-1] = v;                            // Stand guard to keep k<n.
   int k = 0;
   while ( A[k]!=v ) k++;
```

If you prefer to not assume that A[n] exists, use A[n-1] for the sentinel, instead. First, save A[n-1] in a temporary variable, then save the sentinel in A[n-1].

Sentinel Search

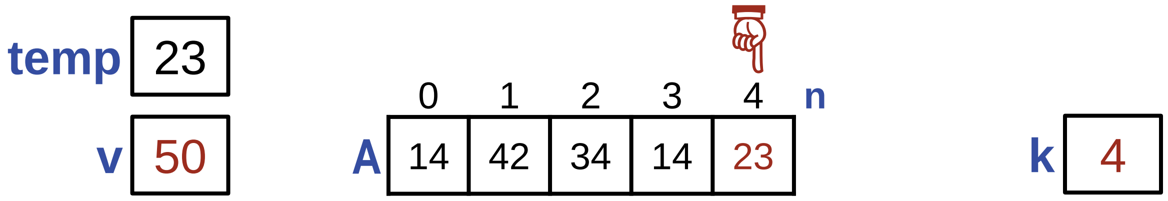


Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int temp = A[n-1];                 // Save A[n-1].
   A[n-1] = v;                        // Stand guard to keep k<n.
   int k = 0;
   while ( A[k]!=v ) k++;
   A[n-1] = temp;                     // Restore A[n-1].
```

If you prefer to not assume that A[n] exists, use A[n-1] for the sentinel, instead. First, save A[n-1] in a temporary variable, then save the sentinel in A[n-1]. After the search, restore A[n-1].

Sentinel Search

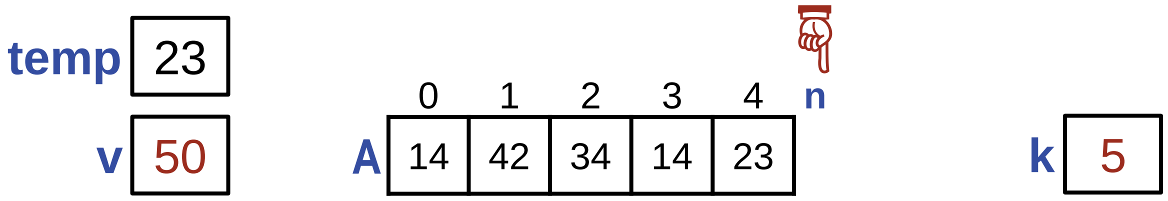


Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int temp = A[n-1];                   // Save A[n-1].
   A[n-1] = v;                          // Stand guard to keep k<n.
   int k = 0;
   while ( A[k]!=v ) k++;
   A[n-1] = temp;                       // Restore A[n-1].
   if ( k==n-1 && A[n-1]!=v ) k=n;      // Test A[n-1] when sentinel is found.
```

If you prefer to not assume that A[n] exists, use A[n-1] for the sentinel, instead. First, save A[n-1] in a temporary variable, then save the sentinel in A[n-1]. After the search, restore A[n-1], and update k, appropriately.

Sentinel Search



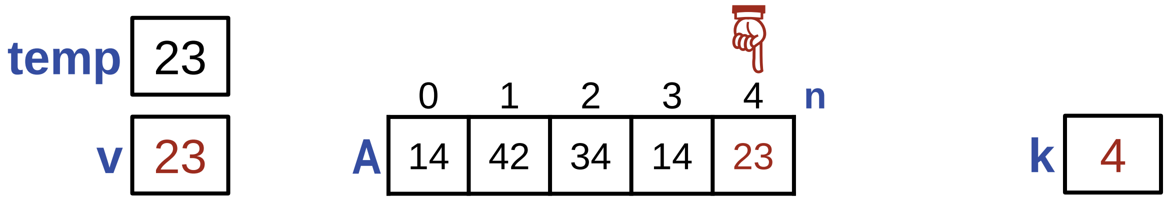
not found

Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int temp = A[n-1];                  // Save A[n-1].
   A[n-1] = v;                         // Stand guard to keep k<n.
   int k = 0;
   while ( A[k]!=v ) k++;
   A[n-1] = temp;                      // Restore A[n-1].
   if ( k==n-1 && A[n-1]!=v ) k=n;     // Test A[n-1] when sentinel is found.
```

If you prefer to not assume that `A[n]` exists, use `A[n-1]` for the sentinel, instead. First, save `A[n-1]` in a temporary variable, then save the sentinel in `A[n-1]`. After the search, restore `A[n-1]`, and update k, appropriately.

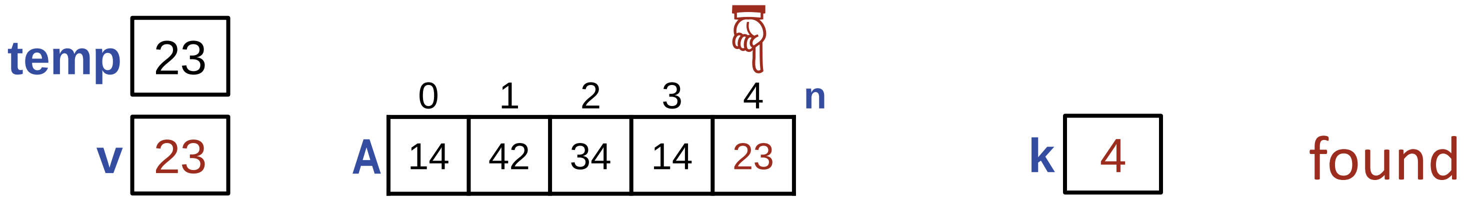
Sentinel Search



Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int temp = A[n-1];                  // Save A[n-1].
   A[n-1] = v;                         // Stand guard to keep k<n.
   int k = 0;
   while ( A[k]!=v ) k++;
   A[n-1] = temp;                      // Restore A[n-1].
   if ( k==n-1 && A[n-1]!=v ) k=n;     // Test A[n-1] when sentinel is found.
```

If you prefer to not assume that `A[n]` exists, use `A[n-1]` for the sentinel, instead. First, save `A[n-1]` in a temporary variable, then save the sentinel in `A[n-1]`. After the search, restore `A[n-1]`, and update k, appropriately.



Technique: Sentinel search.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A. */
   int temp = A[n-1];                 // Save A[n-1].
   A[n-1] = v;                        // Stand guard to keep k<n.
   int k = 0;
   while ( A[k]!=v ) k++;
   A[n-1] = temp;                     // Restore A[n-1].
   if ( k==n-1 && A[n-1]!=v ) k=n;    // Test A[n-1] when sentinel is found.
```

If you prefer to not assume that `A[n]` exists, use `A[n-1]` for the sentinel, instead. First, save `A[n-1]` in a temporary variable, then save the sentinel in `A[n-1]`. After the search, restore `A[n-1]`, and update k, appropriately.

Technique: Sentinel search.

Sentinels have widespread applicability for handling boundary conditions.

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A.
   Assume A[n] exists. */
   A[n] = v;                               // Stand guard to keep k≤n.
   int k = 0;
   while ( A[k]!=v ) k++;
```

Technique: Sentinel search.

Sentinels have widespread applicability for handling boundary conditions, but

```
/* Given array A[0..n-1], n≥0, and value v, let k be the smallest non-negative
   integer s.t. A[k]==v, or let k==n if there are no occurrences of v in A.
   Assume A[n] exists. */
   A[n] = v;                              // Stand guard to keep k≤n.
   int k = 0;
   while ( A[k]!=v ) k++;
```

☞ **Don’t optimize code prematurely.**

New Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   _______
   while ( _______ ) _______
   _______
```

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   _______
   while ( _______ ) _______
   _______
```

A false start.

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   _______
   while ( _______ ) _______
   _______
```

A false start.
Failure to fully understand the problem can prevent starting with a more apt pattern.

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

- ☞ **Analyze first.**
- ☞ **Make sure you understand the problem.**

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

What's a "suffix" in this context?

☞ **Understand the terminology.**

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

What's a "suffix" in this context?

A *suffix* is a sequence of letters at the end of a word.

☞ **Understand the terminology. Reason by analogy.**

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

What's a "suffix" in this context?

A *suffix* is a sequence of letters at the end of a word.

Generalization

A *suffix* is a sequence of _____ at the end of a _____.

☞ **Understand the terminology. Reason by analogy.**

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

What's a "suffix" in this context?

A *suffix* is a sequence of letters at the end of a word.

A *suffix* is a sequence of ______ at the end of a ______.

A suffix is a sequence of array elements at the end of an array.

Generalization

Re-instantiation

☞ **Understand the terminology. Reason by analogy.**

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

What's "descending" in this context?

☞ **Understand the terminology. Reason by analogy.**

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

What's "descending" in this context?

Generalization

Re-instantiation

A *descending* escalator goes down.
A *descending* ________ goes down.
A *descending* sequence of numeric values goes down.

☞ **Understand the terminology. Reason by analogy.**

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

The “longest descending suffix of `A[0..n-1]`“ is a maximally long sequence of elements at the end of the array whose numerical values go down.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Confirm your understanding of a programming problem with concrete examples. Elaborate the expected input/output mapping explicitly.**

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

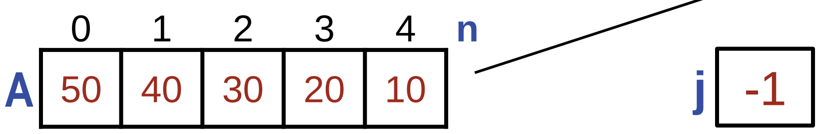
j: 1

Choosing a general example:
The "goldilocks" principle:
Not too long,
not to short,
but just right.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Confirm your understanding of a programming problem with concrete examples. Elaborate the expected input/output mapping explicitly.**



Choosing special-case examples:

"Too long"

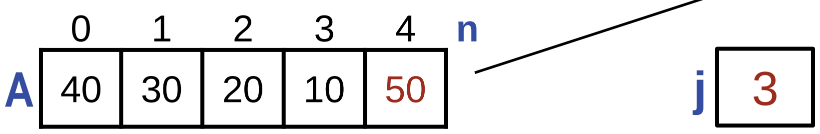
Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Confirm your understanding of a programming problem with concrete examples. Elaborate the expected input/output mapping explicitly.**

Choosing special-case examples:

"Too short"



Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Confirm your understanding of a programming problem with concrete examples. Elaborate the expected input/output mapping explicitly.**

Longest Descending Suffix

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

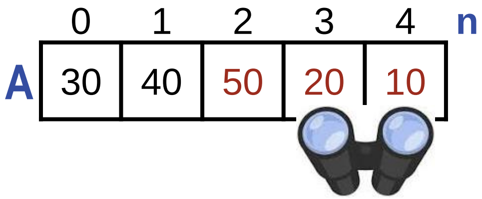
	0	1	2	3	4	n
A	30	40	50	20	10	

Don’t “gestalt” an answer.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

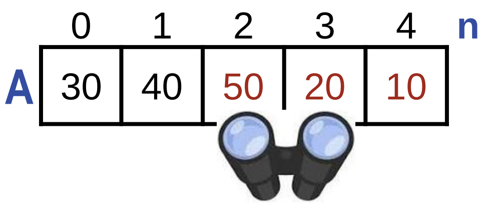


Don't "gestalt" an answer.
Inspect array elements one (or 2) at a time.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

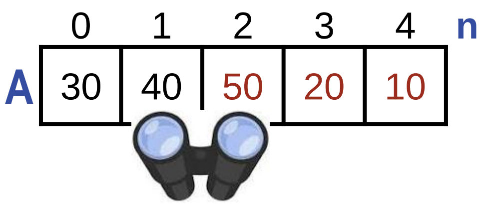


Don't "gestalt" an answer.
Inspect array elements one (or 2) at a time.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

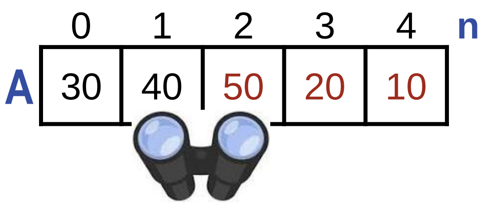


Don't "gestalt" an answer.
Inspect array elements one (or 2) at a time.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

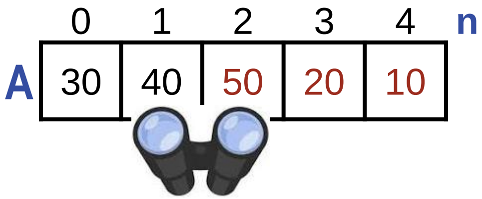


Q. Why did you stop?

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**

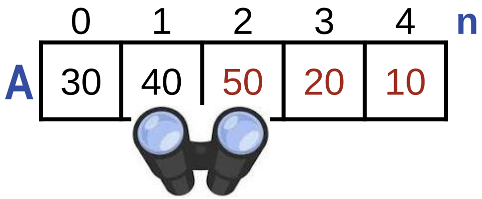


Q. Why did you stop?
A. Because left of pair less than right of pair.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

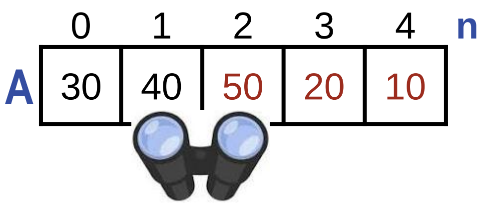


A. Seeking the rightmost pair for which the left element is less than the right element.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your “wetware”. Be introspective. Ask yourself: What am I doing?**



By God, it's a Sequential Search, backward!

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
```

☞ **Seek algorithmic inspiration from experience. Hand-simulate an algorithm that is in your "wetware". Be introspective. Ask yourself: What am I doing?**

Longest Descending Suffix

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

By God, it's a Sequential Search, backward!

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   int j = ______;
   while ( ______ ) j--;
```

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

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   int j = ______;
   while ( A[j]>=A[j+1] ) j--;
```

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

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

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   int j = n-2;
   while ( A[j]>=A[j+1] ) j--;
```

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

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

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

j: -1

"Special case" of a suffix that is the entire array.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   int j = n-2;
   while ( j>=0 && A[j]>=A[j+1] ) j--;
```

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

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

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

j: 3

"Special case" of a suffix of length 1 takes care of itself, as the loop iterates 0 times.

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   int j = n-2;
   while ( j>=0 && A[j]>=A[j+1] ) j--;
```

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

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

Application: Find the Longest Descending Suffix

```
/* Given A[0..n-1], set j so that A[j+1..n-1] is the longest descending
   suffix of A[0..n-1]. */
   int j = n-2;
   while ( j>=0 && A[j]>=A[j+1] ) j--;
```

Q. Why might knowing the longest descending suffix be useful?
A. Think of the elements of `A[0..n-1]` as “letters”, and the array `A[0..n-1]` as a “word”. Consider listing all words that can be made from those letters in lexicographic order, as in a dictionary.

Application: Find the Longest Descending Suffix

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 <u>50</u>
10 20 30 <u>50 40</u>
10 20 40 30 <u>50</u>
10 20 40 <u>50 30</u>
10 20 50 30 <u>40</u>
10 20 <u>50 40 30</u>
10 30 20 40 <u>50</u>
10 30 20 <u>50 40</u>
10 30 40 20 <u>50</u>
etc.

10 20 30 40 <u>50</u>
10 20 30 40 <u>50</u>

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 <u>50</u>
10 20 30 <u>50 40</u>
10 20 40 30 <u>50</u>
10 20 40 <u>50 30</u>
10 20 50 30 <u>40</u>
10 20 <u>50 40 30</u>
10 30 20 40 <u>50</u>
10 30 20 <u>50 40</u>
10 30 40 20 <u>50</u>
etc.

10 20 30 40 <u>50</u>
10 20 30 <u>50 40</u>

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40
10 20 30 50 40

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40
10 20 40 50 30

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50

Application: Find the Longest Descending Suffix

Each transition from one word to the next involves the longest descending suffix. In particular, all words with the corresponding prefix will have been listed, and the next word can be obtained by swapping the last letter of the prefix with the next larger element from the suffix, and reversing the order of the suffix.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
10 20 40 50 30
10 20 50 30 40
10 20 50 40 30
10 30 20 40 50
10 30 20 50 40
10 30 40 20 50
etc.

10 20 30 40 50
10 20 30 50 40
10 20 40 30 50
etc.

Find Minimal

New Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
```

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

POST

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
```

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

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
```

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

Find Minimal

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
```

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

The index k of the minimal element of `A[0..j-1]`.

Find Minimal

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
```

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

Find Minimal

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
   for (int j=____; ____; j++)
      __________
```

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

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
   for (int j=____; ____; j++)
       __________
```

Maintain invariant.

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

Find Minimal

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
   for (int j=____; ____; j++)
      if ( A[j] __ A[k] ) k = ___;
```

Maintain invariant.

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

☞ **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.**

Find Minimal

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
   for (int j=____; ____; j++)
      if ( A[j] __ A[k] ) k = j;
```

Maintain invariant.

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

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

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
   for (int j=____; ____; j++)
      if ( A[j] < A[k] ) k = j;
```

Maintain invariant.

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

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

Find Minimal

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = _____;  // Index of the minimal element of A[0..j-1].
   for (int j=____; j<n; j++)
      if ( A[j] < A[k] ) k = j;
```

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

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

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = 0;      // Index of the minimal element of A[0..j-1].
   for (int j=1; j<n; j++)
      if ( A[j] < A[k] ) k = j;
```

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

Establish invariant.

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

Find Minimal

INVARIANT

Application: Find minimal value in array A[0..n-1].

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1]. */
   int k = 0;      // Index of the minimal element of A[0..j-1].
   for (int j=1; j<n; j++)
      if ( A[j] < A[k] ) k = j;
```

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

The proper behavior is not defined for n=0.

INVARIANT

Application: Find minimal value in array `A[0..n-1]`.

```
/* Given A[0..n-1], find k s.t. A[k] is minimal in A[0..n-1], -1 if n is 0. */
   int k = -1;
   if ( n!=0 ) {
      k = 0;      // Index of the minimal element of A[0..j-1].
      for (int j=1; j<n; j++)
         if ( A[j] < A[k] ) k = j;
      }
```

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

The proper behavior is not defined for n=0.

Precepts used without mention.

- ☞ **Write the representation invariant of an individual variable as an end-of-line comment.**
- ☞ **Termination. Do 2nd. Beware of confusion between condition for continuing and its negation, the condition for terminating. Beware off-by-one errors: stopping one iteration too soon, or one iteration too late. Prevent illegal references using “short-circuit mode” Boolean expressions.**
- ☞ **Initialization. Do 3rd. Initialize variables so that the loop invariant is established prior to the first iteration. Substitute those initial values into the invariant, and bench check the first iteration with respect to that initial instantiation of the invariant.**
- ☞ **Boundary conditions. Dead last, but don’t forget them.**
- ☞ **Find boundary conditions at extrema, and at singularities, e.g., biggest, smallest, 0, edges, etc.**
