cture #6: More Iteration: Sort an Array

out the command-line arguments in lexicographic or-

```
 the quick brown fox jumped over the lazy dog
ox jumped lazy over quick the the
```

```
t {
rint WORDS lexicographically. */
void main(String[] words) {
0, words.length-1);
;


 A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) { /* "TOMORROW" */ }

 one line, separated by blanks. */
int(String[] A) { /* "TOMORROW" */ }
```

How do We Know If It Works?

refers to the testing of individual units (methods) within ather than the whole program.

ing refers to testing of classes or other groupings of data.

ing (or *acceptance testing*) refers to the testing of the of an entire program.

testing is sort of intermediate between unit and system tests that modules work correctly together.

esting refers to testing with the specific goal of check- s, enhancements, or other changes have not introduced ssions).

, we mainly use the JUnit tool for unit testing.

`TestYear.java` in lab #1.

testing is somewhat more *ad hoc*, and customized to . At its simplest, one might just run specific input files program and compare with precomputed outputs.

Test-Driven Development

tests first.

nit at a time, run tests, fix and refactor until it works.

oing to push is fairly lightly in this course, but it is as quite a following.

ot more of it in CS169.

Testing sort

y easy: just give a bunch of arrays to `sort` and then ey each get sorted properly.

e sure we cover the necessary cases:

ses. E.g., empty array, one-element, all elements the

tative "middle" cases. E.g., elements reversed, elements one pair of elements reversed, ...

Simple JUnit

ckage provides some handy tools for unit testing.

notation `@Test` on a method tells the JUnit machinery nethod.

on in Java provides information about a method, class, n be examined within Java itself.)

of methods with names beginning with `assert` then allow ses to check conditions and report failures.

e in the code link for lecture 6.]

Selection Sort

```
s A[L..U], with all others unchanged. */
ort(String[] A, int L, int U) {

*( Index s.t. A[k] is largest in A[L],...,A[U] )*/;
A[k] with A[U] }*/;
tems L to U-1 of A. }*/;
```

Well, OK, not quite.

Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {

dexOfLargest(A, L, U);
[k] with A[U] }*/;
ems L to U-1 of A. }*/;



0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
exOfLargest(String[] V, int i0, int i1) {
```

Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {

dexOfLargest(A, L, U);
[k] with A[U] }*/;
 U-1);      // Sort items L to U-1 of A



0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
exOfLargest(String[] V, int i0, int i1) {
```

Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {

dexOfLargest(A, L, U);
 = A[k];  A[k] = A[U]; A[U] = tmp;
 U-1);      // Sort items L to U-1 of A



0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
exOfLargest(String[] V, int i0, int i1) {
```

Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {

dexOfLargest(A, L, U);
 = A[k];  A[k] = A[U]; A[U] = tmp;
 U-1);      // Sort items L to U-1 of A
```

terative version look like?

Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {

dexOfLargest(A, L, U);
 = A[k];  A[k] = A[U]; A[U] = tmp;
 U-1);      // Sort items L to U-1 of A
```

n:

```
U) {
dexOfLargest(A, L, U);
 = A[k];  A[k] = A[U]; A[U] = tmp;
```

Find Largest

```
0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
exOfLargest(String[] V, int i0, int i1) {
```

Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)

(i0 < i1) */ {
```

Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)

(i0 < i1) */ {
*( index of largest value in V[i0 + 1..i1] )*/;
( whichever of i0 and k has larger value )*/;
```

Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)

(i0 < i1) */ {
ndexOfLargest(V, i0 + 1, i1);
( whichever of i0 and k has larger value )*/;
```

Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)

(i0 < i1) */ {
ndexOfLargest(V, i0 + 1, i1);
[i0].compareTo(V[k]) > 0) ? i0 : k;
i0].compareTo(V[k]) > 0) return i0; else return k;
```

into an iterative version is tricky: not tail recursive.

e arguments to `compareTo` the first time it's called?

Iteratively Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)
;
(i0 < i1) */ {
ndexOfLargest(V, i0 + 1, i1);
[i0].compareTo(V[k]) > 0) ? i0 : k;
i0].compareTo(V[k]) > 0) return i0; else return k;




/ Deepest iteration
 ...?; i ...?)
```

Iteratively Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)
;
(i0 < i1) */ {
ndexOfLargest(V, i0 + 1, i1);
[i0].compareTo(V[k]) > 0) ? i0 : k;
i0].compareTo(V[k]) > 0) return i0; else return k;




// Deepest iteration
 ...?; i ...?)
```

Iteratively Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)
;
(i0 < i1) */ {
dexOfLargest(V, i0 + 1, i1);
[i0].compareTo(V[k]) > 0) ? i0 : k;
0].compareTo(V[k]) > 0) return i0; else return k;



// Deepest iteration
- 1; i >= i0; i -= 1)
.compareTo(V[k]) > 0) ? i : k;
```

Another Problem

f integers, A, of length $N > 0$, find the smallest index, elements at indices $\geq k$ and $< N-1$ are greater than rotate elements k to $N-1$ right by one. For example, s

```
3, 0, 12, 11, 9, 15, 22, 12 }
```

as

```
3, 0, 12, 11, 9, 12, 15, 22 }
```

nple,

```
3, 0, 12, 11, 9, 15, 22, -2 }

 4, 3, 0, 12, 11, 9, 15, 22 }
```

s like this?

```
3, 0, 12, 11, 9, 12, 15, 22 }
```

Your turn

```
Shove {

 elements A[k] to A[A.length-1] one element to the
 where k is the smallest index such that elements
ough A.length-2 are all larger than A[A.length-1].

d moveOver(int[] A) {
 IN
```

Iteratively Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
dexOfLargest(String[] V, int i0, int i1) {
)
;
(i0 < i1) */ {
dexOfLargest(V, i0 + 1, i1);
[i0].compareTo(V[k]) > 0) ? i0 : k;
0].compareTo(V[k]) > 0) return i0; else return k;



// Deepest iteration
- 1; i >= i0; i -= 1)
```

Finally, Printing

```
 one line, separated by blanks. */
int(String[] A) {
 0; i < A.length; i += 1)
.print(A[i] + " ");
rintln();


rovides a simple, specialized syntax for looping
entire array: */
s : A)
.print(s + " ");
```

Another Problem

f integers, A, of length $N > 0$, find the smallest index, elements at indices $\geq k$ and $< N-1$ are greater than rotate elements k to $N-1$ right by one. For example, s

```
3, 0, 12, 11, 9, 15, 22, 12 }
```

as

```
3, 0, 12, 11, 9, 12, 15, 22 }
```

nple,

```
3, 0, 12, 11, 9, 15, 22, -2 }

 4, 3, 0, 12, 11, 9, 15, 22 }
```

s like this?

```
3, 0, 12, 11, 9, 12, 15, 22 }
```

changed. (No, the spec is not ambiguous.)