CS61B Lecture #3

e forgiving during the first week or so, but try to get mitted by Tuesday night. *DBC: Let us know if you can't g to work!*

, there are about 50 people who have accounts but do repositories. You cannot hand anything in without the get this part of the lab done!

n the waiting list, you will not be admitted until you lab (or a space opens up in the one you are waiting on). self from the one you are waiting on and enroll for an risk not getting in.

not be able to enroll until you resolve conflicts with es. We do not encourage signing up for classes with ectures, although there is a way to seek an exception. a final conflict if you have a lecture conflict; we do e that we will have an alternative final at a time you can

Public-Service Announcement

ulting Group is a student-run consulting organiza- rovides strategy consulting services to our clients. tting edge challenges faced exclusively by industry- nology companies in the Silicon Valley. In addition , all of our members go through thoroughly exten- ional development and training programs to become he professional world.... Join our tight-knit fam- form your undergraduate experience through life- ips, networking opportunities, personal mentorships, ources, and more. No prior business or engineering s required.

ormation, please visit our website at vcg.berkeley.edu r table on Sproul! Thank you!"

How do We Know If It Works?

refers to the testing of individual units (methods, classes) ram, rather than the whole program.

, we mainly use the JUnit tool for unit testing.

`TestYear.java` in lab #1.

testing refers to the testing of entire (integrated) set the whole program.

e, we'll look at various ways to run the program against uts and checking the output.

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

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" */ }
```


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 ne pair of elements reversed, ...

Test-Driven Development

tests first.

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

ally going to push it in this course, but it is useful and ollowing.

Simple JUnit

ackage 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.]

Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {
{
( Index s.t. A[k] is largest in A[L],...,A[U] )*/;
[k] with A[U] }*/;
ems 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) {
{
ndexOfLargest(A, L, U);
[k] with A[U] }*/;
 U-1);      // Sort items L to U-1 of A
```


Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {
{
ndexOfLargest(A, L, U);
[k] with A[U] }*/;
ems L to U-1 of A. }*/;
```


Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {
{
ndexOfLargest(A, L, U);
 = A[k];  A[k] = A[U]; A[U] = tmp;
 U-1);      // Sort items L to U-1 of A
```


Selection Sort

```
s A[L..U], with all others unchanged. */
rt(String[] A, int L, int U) {
{
ndexOfLargest(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. */
ort(String[] A, int L, int U) {
{
ndexOfLargest(A, L, U);
 = A[k];  A[k] = A[U]; A[U] = tmp;
 U-1);      // Sort items L to U-1 of A
```

n:

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


Find Largest

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


Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
lexOfLargest(String[] V, int i0, int i1) {
1)

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. */
lexOfLargest(String[] V, int i0, int i1) {
1)

i0 < i1) */ {
```


Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
lexOfLargest(String[] V, int i0, int i1) {
1)

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

into an iterative version is tricky: not tail recursive.

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

Find Largest

```
I0<=k<=I1, such that V[k] is largest element among
 V[I1]. Requires I0<=I1. */
lexOfLargest(String[] V, int i0, int i1) {
1)

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

Iteratively Find Largest

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

of integers, A, of length N, find the smallest index, k,
ements at indices $\geq k$ and $< N$ are greater than A[N].
ments k to $N-1$ right by one. For example, if A starts

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

as

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

mple,

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


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


Iteratively Find Largest

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


roduced a new syntax for the for loop here: */
s : A)
.print(s + " ");
ou like, but let's not stress over it yet! */
```

Your turn

```
Shove {

e 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].

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