Problem. Print out the command-line arguments in lexicographic order:

% java Sort the quick brown fox jumped over the lazy dog
brown dog fox jumped lazy over quick the the

Plan.

```
public class Sort {
    /** Sort and print WORDS lexicographically. */
    public static void main(String[] words) {
        sort(words, 0, words.length-1);
        print(words);
    }

    /** Sort items A[L..U], with all others unchanged. */
    static void sort(String[] A, int L, int U) {
        /* "TOMORROW" */
    }

    /** Print A on one line, separated by blanks. */
    static void print(String[] A) {
        /* "TOMORROW" */
    }
}
```
How do We Know If It Works?

- **Unit testing** refers to the testing of individual units (methods) within a program, rather than the whole program.

- **Module testing** refers to testing of classes or other groupings of methods and data.

- **System testing** (or **acceptance testing**) refers to the testing of the functionality of an entire program.

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

- **Regression testing** refers to testing with the specific goal of checking that fixes, enhancements, or other changes have not introduced faults (regressions).

- In this class, we mainly use the JUnit tool for unit testing.

- Example: AGTestYear.java in lab #1.

- Our system testing is somewhat more *ad hoc*, and customized to the program. At its simplest, one might just run specific input files through the program and compare with precomputed outputs.
Test-Driven Development

• Idea: write tests first.

• Implement unit at a time, run tests, fix and refactor until it works.

• We’re only going to push is fairly lightly in this course, but it is useful and has quite a following.

• You’ll see a lot more of it in CS169.
Testing sort

• This is pretty easy: just give a bunch of arrays to sort and then make sure they each get sorted properly.

• Have to make sure we cover the necessary cases:
  
  - **Corner cases.** E.g., empty array, one-element, all elements the same.
  
  - **Representative “middle” cases.** E.g., elements reversed, elements in order, one pair of elements reversed, ...
Simple JUnit

- The JUnit package provides some handy tools for unit testing.
- The Java annotation `@Test` on a method tells the JUnit machinery to call that method.
- (An *annotation* in Java provides information about a method, class, etc., that can be examined within Java itself.)
- A collection of methods with names beginning with `assert` then allow your test cases to check conditions and report failures.
- [See example in the code link for lecture 6.]
Selection Sort

/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = /*( Index s.t. A[k] is largest in A[L],...,A[U] )*/;
        /*{ Sort items L to U-1 of A. }*/;
    }
}

And we're done! Well, OK, not quite.
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest(A, L, U);
        /*
          * swap A[k] with A[U]
          */
        /*
          * Sort items L to U-1 of A.
          */
    }
}

/** Index k, I0<=k<=I1, such that V[k] is largest element among
 * V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    ...
}
Selection Sort

/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest(A, L, U);
        sort(A, L, U-1);  // Sort items L to U-1 of A
    }
}

/** Index k, I0<=k<=I1, such that V[k] is largest element among
 * V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    ...
}
**Selection Sort**

/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest(A, L, U);
        sort(A, L, U-1); // Sort items L to U-1 of A
    }
}

/** Index k, I0<=k<=I1, such that V[k] is largest element among * V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    ...
}
Selection Sort

/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest(A, L, U);
        sort(A, L, U-1); // Sort items L to U-1 of A
    }
}

What would an iterative version look like?

    while (?) {
        ?
    }
Selection Sort

/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest(A, L, U);
        sort(A, L, U-1); // Sort items L to U-1 of A
    }
}

Iterative version:

while (L < U) {
    int k = indexOfLargest(A, L, U);
    U -= 1;
}
Find Largest

/** Index k, I0<=k<=I1, such that V[k] is largest element among V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    if (?)
        return i1;
    else {
         }
}
/** Index k, I0<=k<=I1, such that V[k] is largest element among * V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    if (i0 >= i1) {
        return i1;
    } else /* if (i0 < i1) */ {

    }
}
/** Index k, I0<=k<=I1, such that V[k] is largest element among
* V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    if (i0 >= i1)
        return i1;
    else /* if (i0 < i1) */ {
        int k = /*( index of largest value in V[i0 + 1..i1] )*/;
        return /*( whichever of i0 and k has larger value )*/;
    }
}
Find Largest

/** Index k, I0<=k<=I1, such that V[k] is largest element among 
 * V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    if (i0 >= i1)
        return i1;
    else /* if (i0 < i1) */ {
        int k = indexOfLargest(V, i0 + 1, i1);
        return /*( whichever of i0 and k has larger value )*/;
    }
}
Find Largest

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

• Turning this into an iterative version is tricky: not tail recursive.
• What are the arguments to compareTo the first time it’s called?
Iteratively Find Largest

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

Iterative:

int i, k;

    k = ?;  // Deepest iteration
for (i = ?; ...?; i ...) 
    k = ?;
return k;
Iteratively Find Largest

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

Iterative:

int i, k;
    k = i1;       // Deepest iteration
for (i = ?; ...?; i ...?)
    k = ?;
return k;
Iteratively Find Largest

/** Value \( k, I_0 \leq k \leq I_1 \), such that \( V[k] \) is largest element among
* \( V[I_0], \ldots, V[I_1] \). Requires \( I_0 \leq I_1 \). */
static int indexOfLargest(String[] V, int i0, int i1) {
    if (i0 >= i1)
        return i1;
    else /* if \((i_0 < i_1)\) */ {
        int k = indexOfLargest(V, i0 + 1, i1);
        return (V[i0].compareTo(V[k]) > 0) ? i0 : k;
        // if \((V[i_0].compareTo(V[k]) > 0)\) return \(i_0\); else return \(k\);
    }
}

Iterative:

int i, k;
\[ k = i1 \];  // Deepest iteration
for (i = i1 - 1; i >= i0; i -= 1)
    \[ k = ? \];
return k;
Iteratively Find Largest

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

Iterative:

int i, k;
 k = i1;    // Deepest iteration
for (i = i1 - 1; i >= i0; i -= 1)
    k = (V[i].compareTo(V[k]) > 0) ? i : k;
return k;
Finally, Printing

/** Print A on one line, separated by blanks. */
static void print(String[] A) {
    for (int i = 0; i < A.length; i += 1)
        System.out.print(A[i] + " ");
    System.out.println();
}

/* Java also provides a simple, specialized syntax for looping
* through an entire array: */
for (String s : A)
    System.out.print(s + " ");
Another Problem

Given an array of integers, $A$, of length $N > 0$, find the smallest index, $k$, such that all elements at indices $\geq k$ and $< N - 1$ are greater than $A[N - 1]$. Then rotate elements $k$ to $N - 1$ right by one. For example, if $A$ starts out as

\[
\{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, 12 \}
\]

then it ends up as

\[
\{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 \}
\]

As another example,

\[
\{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, -2 \}
\]

would become

\[
\{ -2, 1, 9, 4, 3, 0, 12, 11, 9, 15, 22 \}
\]

What if $A$ starts like this?

\[
\{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 \}
\]
Another Problem

Given an array of integers, $A$, of length $N > 0$, find the smallest index, $k$, such that all elements at indices $\geq k$ and $< N - 1$ are greater than $A[N - 1]$. Then rotate elements $k$ to $N - 1$ right by one. For example, if $A$ starts out as

$$\{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, 12 \}$$

then it ends up as

$$\{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 \}$$

As another example,

$$\{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, -2 \}$$

would become

$$\{ -2, 1, 9, 4, 3, 0, 12, 11, 9, 15, 22 \}$$

What if $A$ starts like this?

$$\{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 \}$$

Answer: It’s unchanged. (No, the spec is not ambiguous.)
Your turn

```java
public class Shove {

    /** Rotate elements A[k] to A[A.length-1] one element to the right, where k is the smallest index such that elements k through A.length-2 are all larger than A[A.length-1]. */
    static void moveOver(int[] A) {
        // FILL IN
    }
}
```