# 61B Note 6: SelectionSort and Testing

Kevin Moy

**Abstract**

We'll cover two topics in this note: testing and another sorting algorithm `selectionSort`. Testing is a fundamental skill for *any* programmer. For our second topic, we'll analyze the algorithm of `selectionSort` as well as discuss how to implement it in a modularized and organized manner.

## 1 Testing

Any good programmer MUST be able to **test** their own code- for nobody will (or should) produce perfect code on the first try. Usually we simply test for correctness- but we can also test for things like efficiency or other special cases as well.

**Unit testing** is writing tests for individual units (e.g. methods,classes, etc.) rather than the whole program functionality itself. For example, if we have some function implemented, a unit test could just be testing the functionality or efficiency of that function on various inputs. As far as CS61B projects and homework goes, unit testing will mainly be done through `JUnit`.

**Integration testing** is writing tests for all the (integrated) modules—i.e. testing the functionality of the program as a whole.

Finally, **regression testing** tests that additional enhancements or fixes have not induced additional faults or thrown exceptions.

**Test-driven development** is a system for writing code, based on writing tests *first*. From there, we implement the unit, run tests, and fix as needed.

## 2 JUnit

Java's `JUnit` package is extremely helpful for unit testing. Annotation `@Test` on a method signals to JUnit that it's a like a separate (void) method to run (and test). JUnit also has a bunch of useful `assert` methods that check various equalities in output.

In code, we'd specify these tests in a separate class (perhaps named `UnitTest.java`). Here's some sample code to help:

```
import static org.junit.Assert.* //For assertion statements
import org.junit.Test;

public class UnitTest {
    @Test
    public void testArithmetic(){
        int x = 2;
        int y = 1 + 1;
        assertEquals(x,y);
    }
}
```

In Intellij, an option should appear to run individual `@Test` methods or run them all (independently) in sequence.

## 3 Testing Sorts

Testing sorts is generally fairly easy: given a bunch of (probably unsorted) arrays, ensure that they all are sorted properly by the time the sorting function ends. However, we must also be careful to implement correctness for **edge cases** (special cases): empty arrays, one-element arrays, uniform-element arrays, etc. In other classes, much of project and homework grades are based off of hidden test cases that you must think of and account for yourself in code.

# 4 SelectionSort

Another famous sorting algorithm is `SelectionSort`. This algorithm sorts `arr` through repeatedly finding the `max(arr)` from the unsorted part of the array and placing it at the end (we could also find `min(arr)` repeatedly and add to the beginning). Thus, at all times during the algorithm, we must keep track of two subarrays:

1. Sorted subarray (at end)
2. Unsorted subarray (remaining elements not sorted)

Our gameplan in code:

1. Find the index of the maximum element.
2. Swap that maximum element with the element at the *end* of our sorted subarray (that we built from the end).
3. Recursively selection sort the rest of the unsorted subarray.

```
static void selectionSort(String[] A) {
    return sort(A, 0, A.length-1)
}
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = /*( k | A[k] maximized in A[L:U] )*/;
        /*{ swap A[k] with A[U] }*/;
        /*{ Sort items L to U-1 of A. }*/;
    }
}
```

Now we can go about implementing each step in our gameplan one step at a time. For our first, step, we need to find the index of `arr` that has maximal value. For now, we'll assume we have access to `maxIndex(String[] V, int i0, int i1)` that finds the maximal index among some contiguous subarray of `V`. We'll implement this function in the next section.

Our next step is sending that maximal element to the right place- the beginning of our built subarray, or the end of our unsorted subarray! We know that `U` will contain the right index to swap our maximal element with.

Finally, we must recursively call selection sort on our unsorted subarray, minus the element we just swapped. This is represented by the elements from `L` to `U-1`.

The code implementation for this is below. Ensure you understand how each line of code implements each step of the game plan above.

```
static void selectionSort(String[] A) {
    return sort(A, 0, A.length-1)
}
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    if (L < U) {
        int k = maxIndex(A, L, U);

        String temp = A[k];
        A[k] = A[U];
        A[U] = tmp;

        sort(A, L, U-1);
    }
}
```

# 5 Selection Sort: Iterative

Because we only ever actually change one value (`U`), we can easily make an iterative version of this algorithm:

```
static void selectionSort(String[] A) {
    return sort(A, 0, A.length-1)
}
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
    while (L < U) {
        int k = maxIndex(A, L, U);

        String tmp = A[k];
        A[k] = A[U];
        A[U] = tmp;
```

```
        U--;
    }
}
```

`U` is still the index where we place the next found maximal element in our unsorted subarray. Obviously, as we keep adding maximal elements to our *sorted* subarray, `U` will keep moving backwards.

# 6 MaxIndex: Implementation

We're almost done: we still need to implement `maxIndex`, where we return the index of `max(arr)` for some contiguous subarray whose lower and upper bounds are specified by `i0` and `i1`.

```
static int maxIndex(String[] V, int i0, int i1) {
    if (i0 >= i1)
        return i1;
    else {
        int k = indexOfLargest(V, i0 + 1, i1);
        return (V[i0].compareTo(V[k]) > 0) ? i0 : k;
    }
}
```

This implementation applies a standard recursive idea that the maximum index is either the first index or the maximum index of all indices except the first index. Because we specify a subarray `V[i0...i1]`, however, we stop recursing when `i0 >= i1`, since we've gone past our subarray.

Note that the return statement utilizes a **ternary operator**: if the condition behind the `?` evaluates to `true`, then `i0` is returned, otherwise `k` is returned. This is just a fancy one-liner if-else clause- no problem (at least to my knowledge) ever *requires* the use of a ternary operator.

As is standard in this class, let's find a way to implement `maxIndex` this iteratively as well. This is simply the means of keeping track of two variables: the current maximal value and its index in `V`.

```
static int maxIndexIterative(String[] V, int i0, int i1) {
```

```
    int i, k;
    k = i1; // Deepest iteration. We could also start at i0 and
            //increment.
    for (i = i1 - 1; i >= i0; i -= 1)
        k = (V[i].compareTo(V[k]) > 0) ? i : k ;
    return k;
}
```

Notice we start at the end index `i1`, and simply keep updating our maximal index `k` until we've processed each element in `V[i0...i1]`.

# 7 Extra: Testing SelectionSort

Let's put the two ideas in this note. Now that you have some working code, write some (JUnit) tests to ensure this implementation is correct. For a challenge, try to implement the `selectionSort` variant that repeatedly finds `min(arr)` instead of `max(arr)`.

As a closing note, `selectionSort` is considered a pretty inefficient algorithm because of its **time complexity** of $O(n^2)$. We'll learn more about what this means in a future note; for now, just know that $O(n^2)$ sorting algorithms are sorta lower-tier.