

# Chapter 16

## Searching and Sorting

---

### Hello!

The main topics for this chapter are searching and sorting data. We'll limit our discussion to arrays for searching and sorting, so that our data are linear (it's much simpler to discuss the ideas of searching and sorting, but we could also search and sort a tree structure of data, for instance).

---

### Getting Hyped for Search and Sort

Once we stop doing silly homework assignments and try to do a truly necessary computation with our programming skillz, we often find that we'll apply a single concept a whole lot more—maybe a class structure with 30+ classes, maybe a program with menus inside of menus, maybe reading much more complex files to do computations; in short, we do something a lot. When we get into complex problems that involve other sub-problems, such as sorting results to show to a human, or finding results (as in virtually any time you run a search engine, search for items in some online store, etc.), the store of information available which needs to be operated on can be quite large. So large, in fact, that simple, correct, ways of manipulating data simply aren't feasible. Would you think Google was so great if it took a week to find your search for "Cliff Notes Hemmingway", which you entered in the search box the night before a reading quiz? What if it took minutes to connect a telephone call? Would that be okay? Probably not.

So, although we present some pretty simple algorithms for sorting and searching, keep in mind that we are only showing quite simple sorts; there is quite intensive study into what kind of algorithms perform best in which situations. That said, we still want to be able to manage smaller sets of data, even if our small projects or toy online database of phonebook entries can't be macro-scaled to millions of entries without betraying a bit of naiveté. So let's get sorting, and let's start searching!

---

### Getting Started

Grab the two files, [SearchSortComplete.java](https://cs.gmu.edu/~msnyde14/211/share/SearchSortComplete.java), and [SearchSortIncomplete.java](https://cs.gmu.edu/~msnyde14/211/share/SearchSortIncomplete.java), to your machine and open them (they are also pasted at the end of this document, though cut-pasting from a pdf is dicey).

<https://cs.gmu.edu/~msnyde14/211/share/SearchSortComplete.java>

<https://cs.gmu.edu/~msnyde14/211/share/SearchSortIncomplete.java>

You will be implementing some of the method 'stubs' throughout the chapter in the \*Incomplete version, but you will have the \*Complete version for testing and comparison's sake. This lab will be slightly more about observing code than writing it: we maybe saw some of these examples in class, and we'll just learn about a couple more algorithms today.

There are a few things to notice about this class.

- First of all, scroll down to the bottom and notice there a few 'helper' functions—`printArray`, `getChoice` (two versions), `printMenu`, and `displaySearchResult`. This just helps us to organize our basic testing and exploring harness. In the main method, using these methods makes the code in the main method much shorter, and more descriptive. Instead of seeing a loop, and some `sc.nextFoo()` stuff, we see `getChoice`; instead of a lot of print statements, we see `printMenu`.
  - Notice the two pairs of functions—the two overloads of `binarySearch`, and the two overloads of `quickSort`. The versions with more parameters are called by the others: the simpler ones (fewer parameters) are called "**driver functions**". All they do is take a request, make a start-up call to the 'real' method with any initial values for the additional parameters, and pass any results through. We could just as well have put the extra parameters (usually a zero or an `x.length-1` or something) in our call from the main method, but driver methods are one more way we can abstract details away from our code.
  - Notice the class variable `debug`; We make this variable at the very beginning, and then anywhere we want to print things out in testing mode, we conditionally test if `debug==true`. Taking a quick look inside a method like `bubbleSort` will exemplify this. Maybe using the debugger is a more effective way to view the state, but if you want to take print-style debugging to the next level, this is one approach.
- 

## Basic Searching: Linear Search

A linear search does just that—it starts at the beginning, and searches straight through until it either finds a match or finds itself with no more possible matches left. This is similar to what we've already seen; the method gets (a reference to) an array and a key value to seek, it steps straight through the array until it either finds a match (and returns the index where it was found) or finds out there were no matches (and returns -1 to signify no match found—no valid index is negative).

### Your Turn!

- You should be comfortable writing a linear search algorithm by now, so try implementing the body of `linearSearch` without looking at your notes or the solution.
  - Test your code: noting the default array values created in main, run the program and plug in a key to search for. Search for values that are:
    - in the array
    - not in the array
    - in the array multiple times→ what would be some good unit tests for these? It's a good chance to practice your testing skills if you've got the time.
  - Compare your code against the solution.
- 

## Basic Sorting: Bubble Sort

Bubble Sort runs a double loop. The inner loop compares every single adjacent pairing of two elements (walking down the array: indexes 0 and 1, then 1&2, 2&3, 3&4, etc), and swapping them if they are unsorted relative to each other. Because of the regular pattern of access, no matter where the largest element is, it will get compared (and swapped) with every single smaller value to its right, ensuring that the largest value *has* been placed at the far right.

Each time we perform that swapping-traversal, one more almost-as-large value is guaranteed to have bubbled up to its correct location; if we have an array of length `n`, and we make `(n-1)` traversals, then we know `n`

elements are in order. Thus all the elements are in order. So the outer loop must run as many as  $(n-1)$  times in the worst case.

Bubble Sort is the simplest sort; we might bash it in class, because it does far more swaps than necessary. Notice that, in sorting a particular spot, we might swap many of the values into the sorting spot, if we keep on finding smaller values to the right.

### Your Turn!

- Try implementing bubble sort on your own, without looking at the solution.
  - Using the completed code, turn our debugging print statements on (set the class variable `debug` equal to `true`); run bubble sort and see how it does a lot of swaps. Reset the array, and compare to Insertion Sort (using the complete code's version so that there's an implementation there).
  - Now, try placing different values (and more of them) in the array—specifically, in *decreasing* order. Now see how inefficient bubbleSort can be!
  - **Going Further.** We can make many improvements to the provided bubble sort implementation:
    - Make the inner for-loop smarter: if we've completed  $k$  traversals, we know the last  $k$  elements are sorted; let's end not when  $i$  is less than the array length, but when it has entered the "already sorted" zone.
    - Change the outer for-loop to a while loop. When should it quit? When no swaps were necessary during the last walk. (Use a boolean variable to track this).
- 

## More Efficient Searching: Binary Search

Binary Search could be called the "phonebook" search or the "dictionary" search. It requires the values to be in sorted order. It continually finds the middle of a range that could contain the key (if it is in the array at all), and then either finds the key at that mid-point or calls the method again on the smaller range (to the left or right of the inspected middle location) which is now known to be the only fit location for the key. Binary search relies on the data being sorted – otherwise, it won't work.

### Your Turn!

- **Going Further:** Implement and test your own binary search, trying not to look at the solution (or class notes) in the process. This lends well to a recursive definition, but does not have to be.

### Your Turn!

- Try running this algorithm just like linear search, **without** sorting first. Notice you can get some buggy results—search for a '6' in the array originally listed, and the program crashes!
- A binary search needs sorted data. Let's fix the example:
  - Add a class variable, a boolean named `isSorted`. (This must be static; why?)
  - Go ahead and initialize it to false right there, on the same line. (Reminder: we must always initialize on the same line for class variables—things at the class level don't need an object, so we can't rely on some constructor call to instantiate it).
  - Now, in the `binarySearch` method, at the very beginning, add an if-statement that checks if `isSorted` is false, and if so run a sort algorithm and set `isSorted` to true (take your pick of the various sorting methods in our program).
  - We don't have any other searches in our lab that require a sorted array, so we don't have to fix this anywhere else. However, if we copy over the old array, we need to reset this variable to false. Go to the main method, and look for the option that resets the array to the copy. Immediately following that code, set `isSorted` to false. Now, our `binarySearch` will conditionally sort the information, if necessary, and won't spend all that time if it's already sorted! This is especially good because many

sorting algorithms are pathologically slow when working on mostly- or completely-sorted data. Always sorting first would be an expensive waste of time!

---

## (Slightly) More Efficient Searching: Insertion Sort

Insertion Sort works a bit more like we might actually think about sorting a list. The goal is to first sort a small portion of the list at the front; then, we successively consider one more spot in the list, and keep sliding it leftward (forwards) until it's in sorted place. We now have a slightly longer beginning part of the list that is sorted. We keep performing this action on the next unsorted spot, gradually expanding our sorted portion of the list to consume more and more unsorted elements, until the entire list is sorted.

The first iteration, we make sure the first element by itself is sorted. Well, yes, of course it is. So really, the first step is to add one more unsorted location to our already sorted list of length one, meaning we consider the first two items. If necessary, swap them.

### Your Turn!

- Try implementing insertion sort. Notice that we still will likely have two for-loops: one inner one that handles inserting the next item into our sorted portion of the array, and one outer loop that guides us to adding each unsorted element to the sorted portion, one at a time.

Okay, one way or another, let's now take a look at the solution:

- Notice that the inner loop counts *down* through indexes, starting as high as the outer loop tells us the next unsorted location is. If you had an array drawn on paper, try tracing your finger over indexes that correspond to *j*'s values.
  - After traversing down through our sorted array and swapping our unsorted element as far down as necessary, this additional value is now in its sorted place. Once it no longer needs to be swapped, it won't have to go any further so we can quit the inner loop.
- 

## More Efficient Sorting: Merge Sort

Merge Sort is a different sort of beast; It continually breaks up the array into two halves, sorts both half (using Merge Sort itself), and then *zips* them together—basically, it interleaves the values from the two halves back into one bigger, sorted array.

Merge Sort is a recursive algorithm, as implemented. This means we have to check for base cases first, and then complete the inductive (recursive) cases.

### Your Turn!

- If you're up for the challenge, try implementing merge sort!
  - First, create two sub-lists by splitting the list in half (make two new arrays).
  - Then, recursively call merge sort on each.
  - Then, recreate a list of the original length by always copying in the smallest value from either list that hasn't yet been included. (Relying on their being sorted makes this simple, though you might have a many-else-ifs block of code).

Okay, let's discuss the provided solution. It uses arrays instead of ArrayLists (which Snyder's in-class examples showed), so that you can compare the two implementations.

This sort first creates two halves from the given array. Then, it calls mergeSort on both halves: we now have two smaller sorted arrays, and we just need to combine them into one larger sorted array. Look at the code for that zipping-together portion, and notice it basically checks if the value should come from the next in the first half, the next in the second half, or the smallest of the next in each.

- The reason this works is that lists with only one element in them get returned—they're already sorted—and from this base case, we use the zipping portion to merge two sorted sub-arrays together. So, what this algorithm really does is break the array up into individual spots, then merge them all back together, two arrays at a time, until we get the entire set of values back in one array.
- Note that, especially with the creation of so many arrays all over the place, this is not going to be a very efficient implementation: it uses up a lot of extra space storing all these 'copies' of the array in various levels of separation. An in-place sort (which reuses the current locations in the array instead of creating copies of those values somewhere else) would be much more space-efficient, though it would probably mean more work shuffling values around in that limited space.

### **Your Turn!**

- Try running this sort, as well as in debug mode. It prints a lot, but you can sort of see the two steps—splitting and merging—occurring, just by the width of the left and right arrays being used.
- 

## **One Last Sort: Quick Sort**

Quick Sort functions in an even 'smarter' way. It chooses a pivot value, and then moves anything smaller than it to the left of it, and anything larger than it to the right of it. Since the pivot is now correctly between all values it should be, we then quick-sort the left and right portions with a (recursive) call to Quick Sort again.

### **If you are brave, then it is your turn.**

- Try to implement quickSort. It might also be easiest to use ArrayLists, because the number of values less than (or greater than) the pivot isn't known before you step through the list. You can choose any value in the list as your pivot value; simple choices include the value at the first, middle, or last index. (Choosing the middle one is actually not too bad an uneducated choice). Use recursion to sort the sub-lists to the left and right of the pivot value, once you've placed them all on the correct side of the pivot value (and have figured out where the pivot value belongs, for that matter).

### **Looking at the implementation.**

This sort has a driver function (explained in the beginning of this lab): it builds up the first call to the more-parameters overloaded version of quickSort, which does the real work. Again, this is just to abstract out the extra parameters. Why should we bother to include parameters whose values we can always deduce for the initial call? Also, if we called quickSort with arbitrary alternate values for the left and right, we'd only sort that portion. Partially sorted is close but not particularly useful.

### **Your Turn!**

- Try turning on debugging to see how calls to quickSort operate.
    - What happens when two values are identical?
    - How does quickSort fare when the list is already sorted? When it's sorted the wrong way? (Try starting with values such as {8,7,6,5,4,3,2,1}).
-

# Closing Thoughts on Search & Sort

That's it for searching and sorting for us! Hopefully you now understand what the purpose of searching and sorting is, at least for linear data structures like arrays. In later classes and projects, you will have larger and more complex data structures, but the ideas of searching and sorting will constantly show up: searching through those structures for specific parts (values); trying to organize unsorted data; or, even trying to maintain data in a sorted fashion while adding elements to it one at a time.

We've looked at just a couple of ways to search through data. A linear search was very simple, and needed nothing more than a way to inspect every location in some sequential order. But if we knew we had ordered data, we could do much better with binary search: each inspection allows us to throw out half the remaining search space each time.

We considered a few more ways to sort data, with increasing complexity and different approaches. Bubble sort is very simple, but has pretty poor performance (e.g., it performs many more swaps on average than the other sorts). Insertion sort keeps growing a sorted list at the front of the list, repeatedly sliding the next unsorted value into our sorted portion until it fits. It performs a bit better than bubble sort. Next we considered merge sort, a divide-and-conquer algorithm, which can easily be defined by recursion (but doesn't have to be). It successively splits the list down into sub-lists until they are of size 1 (sorted by definition), and then merges them back together, relying on the simplicity of combining two already-sorted lists. Lastly, we looked at quick sort, which chooses a pivot value, puts all the smaller-than-pivot values on the left, all the larger-than-pivot values on the right, and the pivot between them; it then recursively quicksorts the left and right, and is done.

These versions of sorting had different behaviors – some used lots of space (merge sort), some worked in place (the other three). Some performed better on nearly-sorted data than others.

There are many ways to improve upon the searching and sorting that we've introduced here. What can you find out about these two ubiquitous programming tasks?

# SearchSortIncomplete.java

```
import java.util.*;

public class SearchSortIncomplete {

    public static boolean debug = false;
    public static Scanner sc = new Scanner (System.in);
    public static void main(String[] args) {

        // a value to search for, hoping for its index in the array.
        int key = 0;

        // a resulting index to be returned from a search.
        int result = 0;

        //does the user want to quit?
        boolean quit = false;

        //change this however you like.
        int[] x = { 3, 5, 2, 7, 4, 6, 0, 10, 3 };
        int[] copyOfX = new int[x.length];

        //why can't we just say "copyOfX = x;" ?
        for (int i = 0; i < x.length; i++) {
            copyOfX[i] = x[i];
        }

        while (!quit) {
            printMenu();

            //get a number from 1-9 from the user.
            int choice = getChoice(9);

            //just these options need an extra value, for the key.
            if (choice == 5 || choice == 6) {
                key = getChoice();
            }

            //dispatch the requested operation.
            switch (choice) {
                case 1:
                    bubbleSort(x);
                    break;
                case 2:
                    insertionSort(x);
                    break;
                case 3:
                    mergeSort(x);
                    break;
                case 4:
                    quickSort(x);
                    break;
                case 5:
                    result = linearSearch(key, x);
                    displaySearchResult(result, key);
                    break;
                case 6:
                    result = binarySearch(key, x);
                    displaySearchResult(result, key);
            }
        }
    }
}
```

```

        break;
    case 7:
        printArray(x);
        break;
    case 8:
        for (int i = 0; i < x.length; i++) {
            x[i] = copyOfX[i];
        }
        printArray(x);
        break;
    case 9:
        quit = true;
        break;
    default:
        System.out.println("try again!");
    }
}
}

/*
  Sorting Methods Section.
*/

public static void bubbleSort(int[] a) {
    //TODO: implement!
}

public static void insertionSort(int[] a) {
    //TODO: implement!
}

public static void mergeSort(int[] a) {
    //TODO, for the mighty: implement!
}

public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}

public static void quickSort(int[] a, int leftIndex, int rightIndex) {
    //TODO, for the mighty: implement!
}

/**
 * Searching Methods Section.
 */

public static int linearSearch(int k, int[] a) {
    //TODO: implement!
    return -1;
}

public static int binarySearch(int k, int[] a) {
    return binarySearch(k, a, 0, a.length - 1);
}

private static int binarySearch(int k, int[] a, int left, int right) {
    //TODO: implement!
    return -1;
}

```



```

/**
 * Helpful Functions Section. These all make the main method more
 * "streamlined."
 */

public static void printArray(int[] a) {
    //a slightly smart printing function.
    String vals = "values:";
    String indices = "index: ";
    for (int i = 0; i < a.length; i++) {
        if (a[i] < 10)
            vals += " ";
        if (a[i] < 100)
            vals += " ";
        if (a[i] >= 0)
            vals += " ";
        vals += a[i];

        if (i < 10)
            indices += " ";
        if (i < 100)
            indices += " ";
        indices += " " + i;
    }
    System.out.println(vals + "\n" + indices);
}

public static int getChoice(int num) {
    int x = -1;
    while (x < 1 || x > num) {
        try {
            x = sc.nextInt();
        } catch (Exception e) {
            System.out
                .println("Whoops! try again--enter a menu option from 1 to "
                    + num + ":");
            sc.nextLine();
        }
    }
    return x;
}

public static int getChoice() {
    System.out.println("please enter a number:");
    while (true) {
        try {
            return sc.nextInt();
        } catch (Exception e) {
            System.out.println("Whoops! try again--enter a number:");
            sc.nextLine();
        }
    }
}

public static void printMenu() {
    System.out.println("Please choose:");
    System.out.println("[1]Bubble Sort");
    System.out.println("[2]Insertion Sort");
    System.out.println("[3]Merge Sort");
    System.out.println("[4]Quick Sort");
    System.out.println("[5]Linear Search");
}

```

```
System.out.println("[6]Binary Search");
System.out.println("[7]Print the Array");
System.out.println("[8]Reset the Array");
System.out.println("[9]Quit!");
}

public static void displaySearchResult(int r, int k) {
    //display results for
    if (r != -1) {
        System.out.println("The index for key=" + k + " is " + r);
    } else {
        System.out.println("The key=" + k + " was not found in the array.");
    }
}

}
```

# SearchSortComplete.java

```
import java.util.*;

public class SearchSortComplete {
    public static boolean debug = false;
    public static Scanner sc = new Scanner (System.in);

    public static void main(String[] args) {

        // a value to search for, hoping for its index in the array.
        int key = 0;

        // a resulting index to be returned from a search.
        int result = 0;

        //does the user want to quit?
        boolean quit = false;

        //change this however you like.
        int[] x = { 3, 5, 2, 7, 4, 6, 0, 10, 3 };
        int[] copyOfX = new int[x.length];

        //why can't we just say "copyOfX = x;" ?
        for (int i = 0; i < x.length; i++) {
            copyOfX[i] = x[i];
        }

        while (!quit) {
            printMenu();

            //get a number from 1-9 from the user.
            int choice = getChoice(9);

            //just these options need an extra value, for the key.
            if (choice == 5 || choice == 6) {
                key = getChoice();
            }

            //dispatch the requested operation.
            switch (choice) {
                case 1:
                    bubbleSort(x);
                    break;
                case 2:
                    insertionSort(x);
                    break;
                case 3:
                    mergeSort(x);
                    break;
                case 4:
                    quickSort(x);
                    break;
                case 5:
                    result = linearSearch(key, x);
                    displaySearchResult(result, key);
                    break;
                case 6:
```

```

        result = binarySearch(key, x);
        displaySearchResult(result, key);
        break;
    case 7:
        printArray(x);
        break;
    case 8:
        for (int i = 0; i < x.length; i++) {
            x[i] = copyOfX[i];
        }
        printArray(x);
        break;
    case 9:
        quit = true;
        break;
    default:
        System.out.println("try again!");
    }
}
}

/*
Sorting Methods Section.
*/

public static void bubbleSort(int[] a) {
    int swaps = 0;
    int temp = -1;
    // perform one 'bubbling' pass (n-1) times.
    for (int count = 0; count < a.length-1; count++) {
        for (int i = 0; i < a.length-1; i++) {
            //if these two are out of order, swap them.
            if (a[i] > a[i+1]) {
                if (debug) {
                    System.out.println("swapping "+a[i]+"@" + i + " and " + a[i+1]+" @" + (i+1) +
"].");
                }
                swaps++;
            }
            //swap these locations.
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
        }
    }
    if (debug)
        System.out.println("total swaps: " + swaps);
}

public static void insertionSort(int[] a) {
    int swaps = 0;
    for (int i = 1; i < a.length; i++) {
        for (int j = i; j > 0; j--) {
            // if necessary, swap value at j with value at (j-1).
            if (a[j-1] > a[j]) {
                //before swapping, mention it.
                if (debug) {
                    System.out.println("swapping a[" + j + "-1]=" + a[j-1] + " and a[" + j + "]=" + a[j]);
                    printArray(a);
                    swaps++;
                }
            }
        }
    }
}

```

```

        int temp = a[j];
        a[j] = a[j-1];
        a[j-1] = temp;
    }
}
}
if (debug)
    System.out.println("Number of swaps=" + swaps);
}

public static void mergeSort(int[] a) {

    //base cases
    if (a.length <= 1) {
        return;
    }

    int halfnum = a.length / 2;

    //make two arrays. secondHalf might be one-longer than firstHalf for odd-length arrays.
    int[] firstHalf = new int[halfnum];
    int[] secondHalf = new int[a.length - halfnum];

    //copy in the values to firstHalf and secondHalf.
    //Note the guard statement (i<halfnum), not (i<a.length).
    for (int i = 0; i < halfnum; i++) {
        firstHalf[i] = a[i];
    }
    for (int i=0; i<secondHalf.length;i++){
        secondHalf[i] = a[i + halfnum];
    }

    if (debug) {
        System.out.println("to merge: " + a.length);
        printArray(a);
        printArray(firstHalf);
        printArray(secondHalf);
    }

    //sort the two halves--Recursively!
    mergeSort(firstHalf);
    mergeSort(secondHalf);

    if (debug) {
        System.out.println("having merged: " + a.length);
        printArray(a);
        printArray(firstHalf);
        printArray(secondHalf);
    }

    /**
     * merge the two halves back together. If you think of the smallest
     * cases, this part actually does the sorting!
     */
    int ifst = 0;
    int isnd = 0;

    for (int i = 0; i < a.length; i++) {
        //when there are no more values in the first half.
        if (ifst >= firstHalf.length) {
            a[i] = secondHalf[isnd++];

```

```

    }
    //when there are no more values in the second half.
    else if (isnd >= secondHalf.length) {
        a[i] = firstHalf[ifst++];
    }
    //if both have values left, and the first's next one is smaller.
    else if (firstHalf[ifst] <= secondHalf[isnd]) {
        a[i] = firstHalf[ifst++];
    }
    //if both have values left, and the second's next one is smaller.
    else {
        a[i] = secondHalf[isnd++];
    }
}

if (debug) {
    System.out.println("\tending a length: " + a.length);
    printArray(a);
}

}

public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}

public static void quickSort(int[] a, int leftIndex, int rightIndex) {
    if (debug)
        System.out.println("quicksorting from indexes " + leftIndex + " to " + rightIndex);
    if (leftIndex == rightIndex) {
        return;
    }
    // choose a pivot. we are arbitrarily choosing the middle value,
    // which isn't that better than any other value. But if the data
    // happen to be mostly sorted, this will tend to do all right.
    int pivotIndex = (leftIndex + rightIndex) / 2;
    if (debug) System.out.println("\t with pivot " + pivotIndex);

    //swap pivot value to far left of region.
    int temp = a[leftIndex];
    a[leftIndex] = a[pivotIndex];
    a[pivotIndex] = temp;
    //realign pivot index.
    pivotIndex = leftIndex;

    //partition values around the pivot.
    //i walks right, and pivot swaps smaller values to the left side of it.
    for (int i = leftIndex + 1; i <= rightIndex; i++) {
        if (a[pivotIndex] > a[i]) {
            if (debug)
                System.out.println("\t[swapping " + i + " " + pivotIndex + " " + (pivotIndex + 1));
            //three-way swap. This places the smaller-than-pivot value from a[i]
            // at the old pivot's location, moves the pivot value to the right one,
            // and moves this value displaced by the pivot to where the smaller one
            // happened to be (i.e., anywhere to the right of the pivot).
            temp = a[pivotIndex];
            a[pivotIndex] = a[i];
            a[i] = a[pivotIndex + 1];
            a[pivotIndex + 1] = temp;

            //realign pivot index.
            pivotIndex++;
        }
    }
}

```

```

    }
}

//print out our partially sorted array.
if (debug) {
    System.out.println("pivot now =" + pivotIndex);
    printArray(a);
}

//quickSort the two partitions.
if (pivotIndex > leftIndex) {
    if (debug)
        System.out.println("running left...\n");
    quickSort(a, leftIndex, pivotIndex - 1);
}
if (pivotIndex < rightIndex) {
    if (debug)
        System.out.println("running right...\n");
    quickSort(a, pivotIndex + 1, rightIndex);
}
}

/*
Searching Methods Section.
*/

public static int linearSearch(int k, int[] a) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == k) {
            return i;
        }
    }
    return -1;
}

public static int binarySearch(int k, int[] a) {
    return binarySearch(k, a, 0, a.length - 1);
}

private static int binarySearch(int k, int[] a, int left, int right) {

    int middle = (left + right) / 2;
    if (a[middle] == k) {
        return middle;
    }
    if (left == right) {
        return -1;
    } else if (a[middle] > k) {
        return binarySearch(k, a, left, middle - 1);
    } else {
        return binarySearch(k, a, middle + 1, right);
    }
}

/**
 * Helpful Functions Section. These all make the main method more
 * "streamlined."
 */

public static void printArray(int[] a) {
    //a slightly smart printing function.

```

```

String vals = "values:";
String indices = "index: ";
for (int i = 0; i < a.length; i++) {
    if (a[i] < 10)
        vals += " ";
    if (a[i] < 100)
        vals += " ";
    if (a[i] >= 0)
        vals += " ";
    vals += a[i];

    if (i < 10)
        indices += " ";
    if (i < 100)
        indices += " ";
    indices += " " + i;
}
System.out.println(vals + "\n" + indices);
}

public static int getChoice(int num) {
    int x = -1;
    while (x < 1 || x > num) {
        try {
            x = sc.nextInt();
        } catch (Exception e) {
            System.out
                .println("Whoops! try again--enter a menu option from 1 to "
                    + num + ":");
            sc.nextLine();
        }
    }
    return x;
}

public static int getChoice() {
    System.out.println("please enter a number:");
    while (true) {
        try {
            return sc.nextInt();
        } catch (Exception e) {
            System.out.println("Whoops! try again--enter a number:");
            sc.nextLine();
        }
    }
}

public static void printMenu() {
    System.out.println("Please choose:");
    System.out.println("[1]Bubble Sort");
    System.out.println("[2]Insertion Sort");
    System.out.println("[3]Merge Sort");
    System.out.println("[4]Quick Sort");
    System.out.println("[5]Linear Search");
    System.out.println("[6]Binary Search");
    System.out.println("[7]Print the Array");
    System.out.println("[8]Reset the Array");
    System.out.println("[9]Quit!");
}

public static void displaySearchResult(int r, int k) {

```



```
//display results for
if (r != -1) {
    System.out.println("The index for key=" + k + " is " + r);
} else {
    System.out.println("The key=" + k + " was not found in the array.");
}
}
}
```