Unit 9. Sorting, Searching, and Recursion Notes

Searching. Here are two typical algorithms for searching a collection of items (which for us means an array or a list).

A Linear Search starts at the first element and checks each element in turn until the target value is found. This is easy to implement and the items in the list do not have to be in any particular order.

A Binary Search is designed to work with a collection that is in order. This approach is much faster than a linear search for very large arrays. In order to do a binary search of an array of objects, we must be able to compare the objects.

Here is the code for a binary search:

The array *a* must already be in ascending order.

```
public int binarySearch( int [] a, int target ){
        int left = 0;                           // index of the smallest possible value
        int right = a.length - 1;               // index of the largest possible value

        while ( left <= right ) {
                int middle = ( right + left ) / 2;          // guess the middle index

                if ( a[ middle ] == target )
                        return middle;

                else if ( a[ middle ] < target )

                        ______________________

                else

                        ______________________
        }

        return -1;
}
```

You found it. Return the index of the target

The value at middle is smaller than target; move the left side in.

The value at middle is larger than target; move the right side in.

Every time the target value is not found, either left becomes bigger or right becomes smaller. If the target is not in the array, then eventually left becomes bigger than right and we return -1.

The above code can work for any primitive data type but we cannot use >, >=, < or <= to compare two object references. The following code:

```
if ( "a" > "b" )
        System.out.println( "Is this ok?" );
```

generates a compiler error: operator > cannot be applied to java.lang.String, java.lang.String

If we want to sort and do a binary search of a collection of objects then we need a way to compare objects and say that one is less than or greater than another.

The comparable interface has just one method, compareTo which is defined as follows:

public int compareTo(Object other)

Usage of compareTo	**Value Returned**
obj1.compareTo(obj2)	
obj1.compareTo(obj2)	
obj1.compareTo(obj2)	

The String class implements the Comparable interface. Strings are compared character by character. Each character is encoded and stored in binary.

The characters A to Z are stored as	0100 0001 (65) to 0101 1010 (90)
The characters a to z are stored as	0110 0001 (97) to 0111 1010 (122)
The digits 0 to 9 are stored as	0011 0000 (48) to 0011 1001 (57)

If the first two characters are the same, then the next two are compared and so on.

If s1 is “apple” and s2 is “banana” then what is displayed?

a) apple is greater
b) banana is greater
c) equal

If s1 is “4” and s2 is “2000” then what is displayed?

a) 4 is greater
b) 2000 is greater
c) equal

If s1 is “hey” and s2 is “HEY” then what is displayed?

a) hey is greater
b) HEY is greater
c) equal

If s1 is “w” and s2 is “www” then what is displayed?

a) w is greater
b) www is greater
c) equal

```
String s1, s2;
// s1 and s2 are assigned values

int result = s1.compareTo( s2 );
if ( result > 0 )
        System.out.println( s1 + " is greater");
else if ( result < 0 )
        System.out.println( s2 + " is greater" );
else
        System.out.println( "equal" );
```

The following outline shows how the compareTo method can be implemented.

```
public class SomeClass implements Comparable {
        public int compareTo( Object other ) {
                if ( other instanceof SomeClass == false )
                        throw new IllegalArgumentException("other must be SomeClass object");

                SomeClass sc = (SomeClass) other;
                // write code that compares this with sc
                // return a zero if this equals sc
                // return a negative number if this is less than sc
                // return a positive number if this is greater than sc
        }
}
```

Objects are compared to each other based on the values of all or some of their instance variables.

For example, if you had a Student class that contained fields for student id and GPA. Your code may compare students just on the basis of their names or just based on their grades or some approach that combines the two.

IMPORTANT.
(1) The compareTo method should return zero under the same circumstances that the equals method returns true.
(2) The compareTo method should return a non-zero value under the same circumstances that the equals method returns false.

Sorting. There are many algorithms for sorting a collection of items. The AP curriculum requires that we cover 3 algorithms; two we discuss now and one we will discuss after we cover recursion. These are written from the perspective of putting items into ascending order those they can easily be modified to arrange items in descending order.

Selection Sort. Here’s a brief outline of how a selection sort works.

```
public static void selectionSort( int [] a ){
        for ( int n = 0; n < a.length - 1; n++ ){
                // 1. Search the rest of the array (from n+1  to the end) for the smallest value
                // 2. If that value is less than the value at index n, switch the values.
        }
}
```

Here's an example of how the selection sort algorithm works in practice

Index	**Values in the Array**	**After 1 Iteration**	**After 2 Iterations**	**After 3 Iterations**	**After 4 Iterations**
0	45				
1	4				
2	6				
3	33				
4	18				

Insertion Sort. Here is the code for the insertion sort algorithm for an array of ints. Most of the work is done by the helper method

```
public static void insertionSort( int [] a ){
        for ( int n = 1; n < a.length; n++ ){
                insertAbove( a, n );
        }
}

// precondition: a is sorted (ascending) from index zero to n-1 AND 0 < n < a.length
// insert the value at n into the proper index and shift other values as needed
// postcondition: a is sorted from index zero to n
private static void insertAbove( int [] a, int n ){
        while ( n > 0 ){
                if ( a[n-1] <= a[n] )
                        return;
                else {
                        switchValues( a, n -1, n );
                        n--;
                }
        }
}

private static void switchValues( int [] a, int n1, int n2 ){
        int temp = a[n1];
        a[n1] = a[n2];
        a[n2] = temp;
}
```

Here's an example of how the insertion sort algorithm works in practice

Index	**Values in the Array**	**After 1 Iteration**	**After 2 Iterations**	**After 3 Iterations**	**After 4 Iterations**
0	45				
1	4				
2	6				
3	33				
4	18				

Recursion and Recursive Methods. You are only required to be able to read and evaluate the result of a recursive method.

A recursive method is ____

Any algorithm that can be implemented using an iterative approach can be implemented using a recursive approach and vice versa. Sometimes recursion provides the clearest, shortest, and most elegant solution to a programming problem; and sometimes iteration is the best approach.

Each recursive call to a method creates new local variables and parameter.

Guidelines for Recursive Methods.

- A recursive method must ____

  ____
- The recursive step, in which the method calls itself, must ______________________________

  ____

Example 1.
 	
int x = count(4);
 	
What is x?
	```
	public int count( int n ) {
	        if ( n < 1 )
	                return 0;
	        else
	                return n + count( n - 1 );
	}
	```

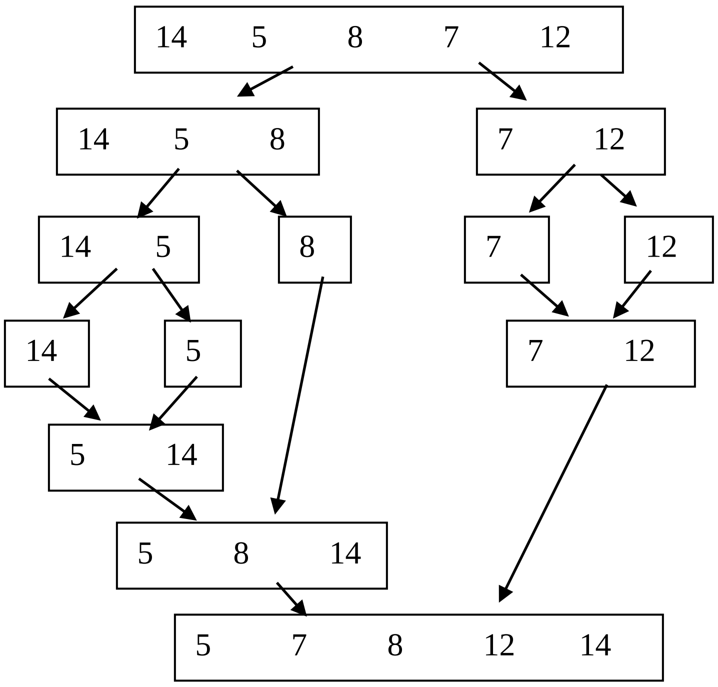
Example 2.
 	
r1("glad");
 	
What is displayed?
	```
	public void r1( String s ){
	        if ( s.length() == 0 )
	                return;

	        System.out.println( s );
	        r1( s.substring( 1 ) );
	}
	```

The MergeSort Algorithm is a recursive sorting algorithm that has a "divide and conquer" approach to sorting an algorithm.

Here's the basic approach:

Keep "dividing" the array into smaller and smaller sections until each section consists of a single element. Then begin merging the sections today. At each step, you will be merging two sequential sections that are individually sorted into one larger sorted section.

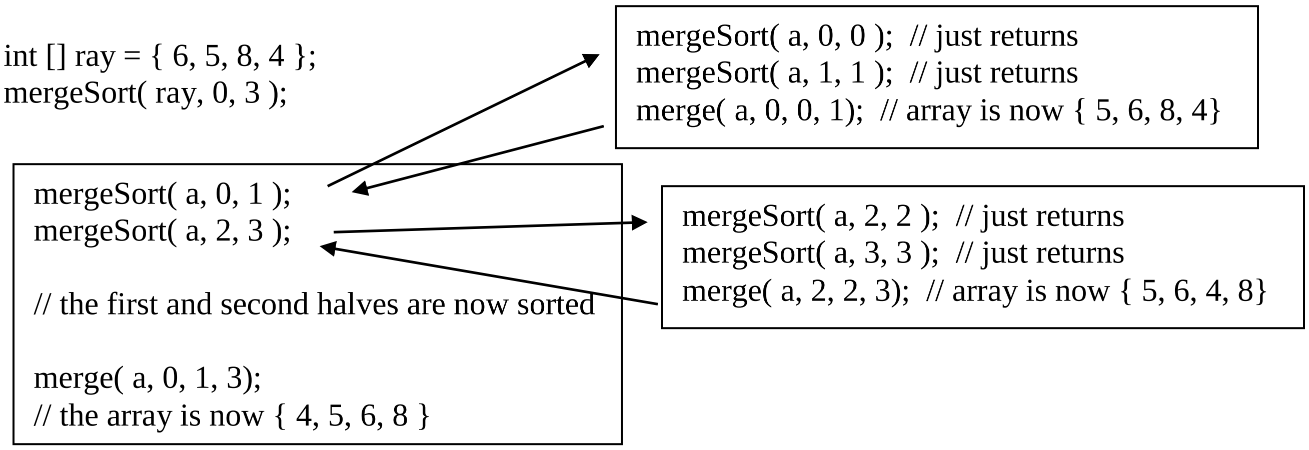
IMPORTANT. Generally speaking, a merge sort is much faster than either a selection or insertion sort.



```
// Precondition: 0 <= start <= end < a.length
// Postcondition: a[] is sorted in ascending order

public void mergeSort (int [] a, int start, int end ) {
        if ( start == end )
                return;

        int middle = (start + end)/2;
        mergeSort( a, start, middle );
        mergeSort( a, middle + 1, end  );
        merge( a, start, middle, end );
}
```

Here's the textbook's implementation. The merge method weaves the two sorted sections into a larger sorted section.

Here is an example of the mergeSort algorithm.



```
int [] ray = { 6, 5, 8, 4 };
mergeSort( ray, 0, 3 );
```

```
mergeSort( a, 0, 1 );
mergeSort( a, 2, 3 );

// the first and second halves are now sorted

merge( a, 0, 1, 3);
// the array is now { 4, 5, 6, 8 }
```

```
mergeSort( a, 0, 0 );  // just returns
mergeSort( a, 1, 1 );  // just returns
merge( a, 0, 0, 1);  // array is now { 5, 6, 8, 4}
```

```
mergeSort( a, 2, 2 );  // just returns
mergeSort( a, 3, 3 );  // just returns
merge( a, 2, 2, 3);  // array is now { 5, 6, 4, 8}
```