

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. *(in fact if not stated assume not sorted)*

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 )
            left = middle + 1;
        else
            right = middle - 1;
    }
    return -1;
}
```

makes sense as we have already examined the element @ index middle

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)	neg number if obj1 comes before obj2 ie
obj1.compareTo(obj2)	pos number if obj1 comes after obj2
obj1.compareTo(obj2)	0 if obj1 is equal to obj2

"a" before "c"

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)

phone book
dictionary
iphone

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

<p>If s1 is "apple" and s2 is "banana" then what is displayed?</p> <p>a) apple is greater <input checked="" type="radio"/> b) banana is greater c) equal</p> <p>If s1 is "4" and s2 is "2000" then what is displayed?</p> <p>a) 4 is greater <input checked="" type="radio"/> b) 2000 is greater c) equal</p> <p>If s1 is "hey" and s2 is "HEY" then what is displayed?</p> <p>a) hey is greater <input checked="" type="radio"/> b) HEY is greater c) equal</p> <p>If s1 is "w" and s2 is "www" then what is displayed?</p> <p>a) w is greater <input checked="" type="radio"/> b) www is greater c) equal</p>	<pre>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");</pre>
---	---

* numerics < caps < lowercase

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.
    }
}
```

Sorts on unit 9 resources

Selection sort starts at 0 or end of array.

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	<u>4</u>	4	4	4
1	4	45	<u>6</u>	6	6
2	6	6	45	<u>18</u>	18
3	33	33	33	33	<u>33</u>
4	18	18	18	45	45

$n-1$ iterations through outer loop
 $O(n^2)$

↑
 note: no change in array
 but iteration occurs!

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;
}
```

great example of
procedural abstraction

note: insertion sort starts @ 1 or 1 from end.
 easy to spot difference between code segments

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	4	4	4	4
1	4	45	6	6	6
2	6	6	45	33	68
3	33	33	33	45	33
4	18	18	18	18	45

★ $n-1$ outer loop iterations. $O(n^2)$ on a sorted array $O(n)$
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 a method which calls itself to solve a problem

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.

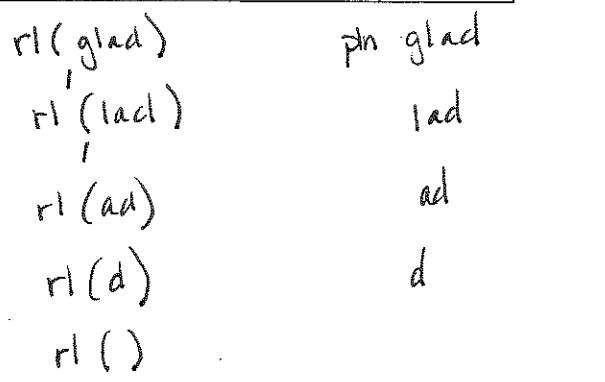
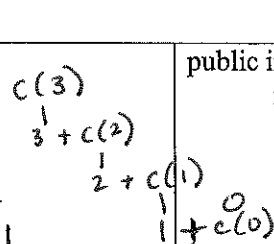
Guidelines for Recursive Methods.

- A recursive method must have an end condition (similar to loops)
this is the condition under which it stops calling itself
- The recursive step, in which the method calls itself, must create
local parameters (ie: new version of call)

stack method

c(0)	0 +
c(1)	1 +
c(2)	2 +
c(3)	3 +
c(4)	4 +

<p>Example 1.</p> <p>int x = count(4);</p> <p>What is x? 10</p>	<p>public int count(int n) { if (n < 1) return 0; else return n + count(n - 1); }</p> <p>stop condition new version of call</p>
<p>Example 2.</p> <p>r1("glad");</p> <p>What is displayed?</p>	<p>public void r1(String s) { if (s.length() == 0) return; System.out.println(s); r1(s.substring(1)); }</p> <p>what is printed</p>



Similar to binary search
in each time

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

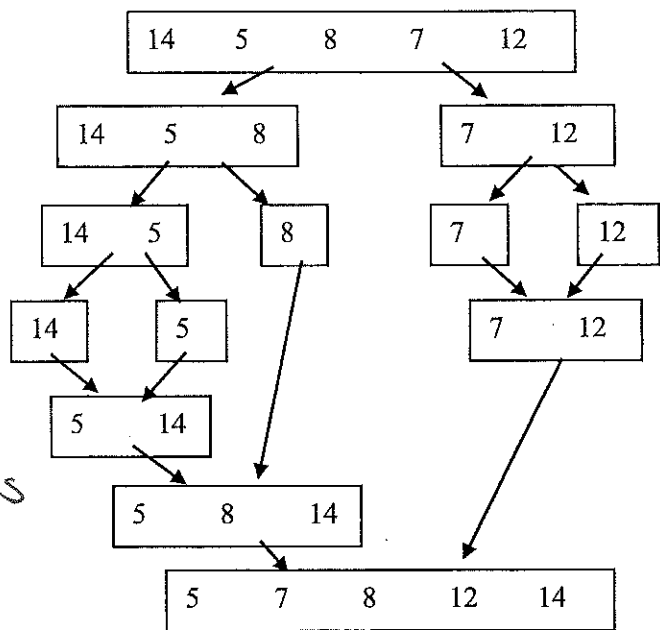
Here's the basic approach:

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.

O(n log n) for best & worst cases

// 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 );
}
  
```

even if named funny, merge sort has to recurse & in two parts left & right halves!

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 }
  
```