

Chapter 1101: Searching & Sorting

```
DEFINE HALFHEARTEDMERGESORT(LIST):  
  IF LENGTH(LIST) < 2:  
    RETURN LIST  
  PIVOT = INT(LENGTH(LIST) / 2)  
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])  
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])  
  // UMMMMM  
  RETURN[A,B] // HERE. SORRY.
```

WEEK 5: INCLUDING A DISCUSSION OF
ALGORITHMIC COMPLEXITY AND RUNNING TIMES

Searching & Sorting using the Java Class Libraries

- To determine if your data contains a particular value, you need to search the data
- When the data are stored, **unsorted**, in a list, this requires sequentially searching the elements in the list until the value of interest is found or until all the elements have been examined.
- The *indexOf* method performs such a sequential search.
- If your data are stored in a **sorted** list, there are more efficient search techniques.

Binary Search

- Much faster than searching elements sequentially.
- Both the Java Arrays class and Java list classes have static binary search methods.
- The method examines the value that is located at the midpoint of the sorted data elements, then searches further **ONLY** in the appropriate half, again targeting the midpoint of the selected range.
- Can narrow in on the specified value quickly or determine that it is not contained in the data set.

Examples:

- Search middle value first
- Determine if target is greater than or less than the value at the midpoint
- If the target is less than the midpoint value, search the left half of the data
- Otherwise, search the right half
- Keep repeating this technique until the target value is found or until it is clear that it is not contained in the data

Search for 2

0	1	2	4	5	6	7	8	9
0	1	2	4					
		2	4					

Search for 1

0	1	2	4	5	6	7	8	9
0	1	2	4					

Search for 9

0	1	2	4	5	6	7	8	9
					6	7	8	9
							8	9
								9

Search for 3

0	1	2	4	5	6	7	8	9
0	1	2	4					
		2	4					
			4					
				END				

Using binarySearch

```
// binary search on an array
int[] numbers = {-3, 2, 8, 12, 17, 29, 44, 58, 79};
int index = Arrays.binarySearch(numbers, 29);
System.out.println("29 is found at index " + index + " in the array.");

// binary search on an ArrayList
ArrayList<Integer> list = new ArrayList<>();
list.addAll(Arrays.asList(-3, 2, 8, 12, 17, 29, 44, 58, 79));
int indexAL = Collections.binarySearch(list, 29);
System.out.println("29 is found at index " + indexAL + " in the ArrayList.");
```

Problems @ Javadoc Declaration Console

<terminated> Test [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

29 is found at index 5 in the array.

29 is found at index 5 in the ArrayList.

Note: Since ArrayList is part of the Java Collections Framework, call the static method Collections.binarySearch, rather than ArrayList.binarySearch.

binarySearch only guaranteed to work on SORTED data

```
// binary search on an array
int[] numbers = {-3, 2, 5, 8, 12, 17, 92, 29, 44, 58, 4, 79};
int index = Arrays.binarySearch(numbers, 29);
System.out.println("29 is found at index " + index + " in the array.");

// binary search on an ArrayList
ArrayList<Integer> list = new ArrayList<>();
list.addAll(Arrays.asList(-3, 2, 5, 8, 12, 17, 92, 29, 44, 58, 4, 79));
int indexAL = Collections.binarySearch(list, 29);
System.out.println("29 is found at index " + indexAL + " in the ArrayList.");
```

Problems @ Javadoc Declaration Console

<terminated> Test [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

29 is found at index -7 in the array.

29 is found at index -7 in the ArrayList.

If binarySearch is called on unsorted data, the results are undefined and the algorithm doesn't guarantee that it will return the right answer.

Sorting

- The Java class libraries provide sorting methods for arrays and lists.
- Arrays must be of *types that can be compared* – either primitives or objects that implement the **Comparable** interface (from Ch. 10).
- Use the *Arrays.sort* method to sort arrays
 - The “**quicksort**” method is used to sort primitives
 - The “**mergesort**” method is used for object data
- For lists, use the *Collections.sort* method.
 - Uses “**mergesort**” for object data.

Custom Ordering with Comparators

- Sometimes you'll want to search or sort a collection of object in a way that differs from the way specified by the collections **Comparable** implementation.
- In such situations, you can define an *external comparison function* with an object called a **comparator**.
- *Examples:*
 - Comparing strings without considering case differences
 - Comparing points using the x-coordinates first when the *compareTo* method compares y-coordinates first.

Using a comparator to compare strings without considering case

- Let's look at a comparator included in the Java class libraries: `CASE_INSENSITIVE_ORDER`

```
// sort strings using case-insensitive comparator
String[] strings = {"Foxtrot", "alpha", "echo", "golf", "bravo", "hotel", "Charlie", "DELTA"};
Arrays.sort(strings, String.CASE_INSENSITIVE_ORDER);
System.out.println(Arrays.toString(strings));
```

Problems @ Javadoc Declaration Console

<terminated> Test [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

[alpha, bravo, Charlie, DELTA, echo, Foxtrot, golf, hotel]

- Using the `Arrays.sort` method would have produced:

[Charlie, DELTA, Foxtrot, alpha, bravo, echo, golf, hotel]

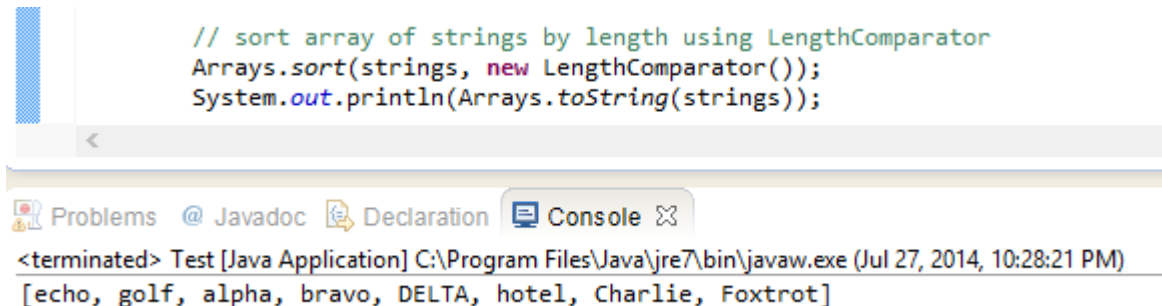
Making a new Comparator

- What if you wanted to sort strings based on string length?
- You could write a new **Comparator** to do this:

```
public class LengthComparator implements Comparator<String>{  
    public int compare(String s1, String s2){  
        return s1.length() - s2.length();  
    }  
}
```

- Then, call *Arrays.sort* using this compare method:

```
// sort array of strings by length using LengthComparator  
Arrays.sort(strings, new LengthComparator());  
System.out.println(Arrays.toString(strings));
```



<terminated> Test [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jul 27, 2014, 10:28:21 PM)
[echo, golf, alpha, bravo, DELTA, hotel, Charlie, Foxtrot]

Useful Comparators & Methods

Comparator/Method	Description
<code>Arrays.binarySearch(array, value, comparator)</code>	Returns the index of the given value in the given array, assuming the array is sorted, or returns a negative if the value not found
<code>Arrays.sort(array, comparator)</code>	Sorts the given array in the ordering of the given comparator.
<code>Collections.binarySearch(list, value, comparator)</code>	Returns the index of the given value in the given (sorted) list , or returns a negative number if the value not found
<code>Collections.max(collection, comparator)</code> & <code>Collections.min(collection, comparator)</code>	Returns the largest or smallest value in the collection (respectively, according to the ordering of the given comparator
<code>Collections.reverseOrder()</code> & <code>Collections.reverseOrder(comparator)</code>	Returns a comparator that compares objects in the opposite of their natural order or the order of the given comparator
<code>Collections.sort(list, comparator)</code>	Sorts the given list in the ordering of the given comparator.
<code>String.CASE_INSENSITIVE_ORDER</code>	Sorts strings alphabetically, ignoring capitalization.

Implementing Searching & Sorting Algorithms: *Sequential Search*

- Perhaps the simplest way to search an array
- Loop over each element, checking each individually
- You are all familiar with this technique.

```
System.out.println(Arrays.toString(numbers));
int index5 = indexOf(numbers, 5);
System.out.println("The value 5 is found at index: " + index5);
}

// sequential search algorithm
// Returns the index at which a given target number first appears,
// or -1 if it is not found
public static int indexOf(int[] intList, int target){
    for(int i = 0; i < intList.length; i++){
        if (intList[i] == target){
            return i;
        }
    }
    return -1; // not found
}

<
Problems @ Javadoc Declaration Console
<terminated> Test [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
[-3, 2, 5, 8, 12, 17, 29, 44, 58, 79]
The value 5 is found at index: 2
```

Implementing Searching & Sorting Algorithms: *Binary Search*

- As described in an earlier slide – takes advantage of the ordering of an array
- Keeps track of the indices of the section of interest in the array (initially the starting and ending indices of the array)
- The algorithm repeatedly examines the center element in the range of interest, eliminating either the smaller or larger half depending on the comparison done between the center element's value and the target value

Binary search algorithm

```
// Binary search algorithm
// Returns an index at which the target
// appears in the given input array, or
// -1 if not found
// pre: the array is sorted
public static int binarySearch(int[] numbers, int target){
    int min = 0;
    int max = numbers.length - 1;
    while (min <= max){
        int mid = (max + min) / 2;
        if (numbers[mid] == target){
            return mid; // found it!
        } else if (numbers[mid] < target){
            min = mid + 1; // too small
        } else { // numbers[mid] > target
            max = mid - 1; // too large
        }
    }
    return -1; // not found
}
```

Adding print statements to the binarySearch method to see how it's working gives the following output (target = 5):

```
[ -3 2 5 8 12 17 29 44 58 79 ]
The middle value is: 12 at index: 4
-----
Min Index = 0, Max Index = 3
[ -3 2 5 8 ]
The middle value is: 2 at index: 1
-----
Min Index = 2, Max Index = 3
[ 5 8 ]
The middle value is: 5 at index: 2
-----
The value 5 is found at index: 2
```

NOTE: Integer division is used in the calculation to find the midpoint.

Implementing Searching & Sorting Algorithms: *Recursive Binary Search*

- We just looked at an iterative implementation of `binarySearch` that used a while loop
- `binarySearch` can also be implemented recursively
- The recursive method accepts the min and max indices as parameters
- On each pass of the recursive algorithm, the code recursively examines the middle element
- Based on the comparison of this element with the target, the code determines which side of the array to examine
- This process repeats until the target is located or the whole array has been eliminated from consideration

Recursive binary search algorithm

```
// Recursive binary search algorithm
// Returns an index at which the target
// appears in the given input array, or
// -1 is not found
// pre: the array is sorted
private static int binarySearchR(int[] numbers, int target, int min, int max){
    //base case
    if (min > max){
        return -1; // not found
    } else{
        // recursive case
        int mid = (max + min) / 2;
        if(numbers[mid] == target){
            return mid;
        } else if (numbers[mid] < target){
            return binarySearchR(numbers, target, mid + 1, max);
        } else {
            return binarySearchR(numbers, target, min, mid -1);
        }
    }
}
```

NOTE:

The adjustment to the indices is made when they are recursively passed into the method

Searching objects (instead of primitives)

- Remember that relational operators don't work on objects
- Instead of “**==**” → use the ***equals*** method
- Instead of “**<**” or “**>**” → use the ***compareTo*** method
- See pp. 846-847 if you need a refresher on how to use these methods

Selection Sort

This sorting algorithm makes **many** passes over an array

During each pass, the smallest remaining element is selected and placed in its proper place near the front of the array.

```
// Places the elements of the given array into sorted order
// using the selection sort algorithm.
// post: array is in sorted (nondecreasing) order
public static void selectionSort(int[] a){
    for(int i = 0; i < a.length - 1; i++){
        // find the index of the smallest element
        int smallest = i;
        for(int j = i + 1; j < a.length; j++){
            if(a[j] < a[smallest]){
                smallest = j;
            }
        }
        swap(a, i, smallest); // swap the smallest to the front
    }
}

public static void swap(int[] list, int i, int j){
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}
```

Problems @ Javadoc Declaration Console

<terminated> Test [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

Array before sorting: [25, 24, 32, 95, 84, 80, 64, 9, 15, 89]
Array after sorting: [9, 15, 24, 25, 32, 64, 80, 84, 89, 95]

Selection Sort - mechanics

Array before sorting: [47, 26, 77, 20, 64]

Before swap: [47, 26, 77, 20, 64]

Swap: 20 and 47

After swap: [20, 26, 77, 47, 64]

Before swap: [20, 26, 77, 47, 64]

Swap: 26 and 20

After swap: [20, 26, 77, 47, 64]

Before swap: [20, 26, 77, 47, 64]

Swap: 47 and 77

After swap: [20, 26, 47, 77, 64]

Before swap: [20, 26, 47, 77, 64]

Swap: 64 and 77

After swap: [20, 26, 47, 64, 77]

Array after sorting: [20, 26, 47, 64, 77]

NOTE:

When the element being is examined is in the correct position, the swap method is still called.

A sequential search is used in each pass to locate the smallest element.

As the method progresses, the sorted (left) side of the array increases in length, and the unsorted (right) side correspondingly decreases in length.

Merge Sort

- Algorithms like selection sort are **impractically slow** when the number of elements being sorted reaches the tens of thousands
- The ***merge sort*** algorithm is named for the observation that if you have **two sorted subarrays**, these can easily be ***merged*** into a sorted array.
- **The basic idea:**
 1. Split the array into two halves.
 2. Sort the left half.
 3. Sort the right half.
 4. Merge the two halves.

Merge Sort, continued...

Step 1 and Step 4 are relatively straightforward to implement.

Notice that the `if` statement in the merge method is checking to ensure that there are still elements in the right and left arrays before taking a value from them.

If index `iR` is greater than the length of the right array, elements will be taken from the left array.

Similar checks exist to ensure items are only taken from the left array when it is legal to do so.

```
public static void mergeSort(int[] a){
    // 1. Split array into two halves.
    int[] left = Arrays.copyOfRange(a, 0, a.length / 2);
    int[] right = Arrays.copyOfRange(a, a.length / 2, a.length);

    // 2. and 3. Sort the two halves.

    // 4. Merge the two (sorted) halves.
    merge(a, left, right);
}

//Merges the left and right arrays into the given result array.
// pre: result is empty; left/right are sorted
// post: result contains result of merging sorted lists
public static void merge(int[] result, int[] left, int[] right){
    int iL = 0; // index into the left array (L)
    int iR = 0; // index into the right array (R)
    for(int i = 0; i < result.length; i++){
        if(iR >= right.length || (iL < left.length && left[iL] <= right[iR])){
            result[i] = left[iL]; // take from the left
            iL++;
        } else {
            result[i] = right[iR]; // take from the right
            iR++;
        }
    }
}
```

Merge Sort, continued

- We still need to **sort** the two halves of the array.
- Any sorting method could be used here, but if a method like selection sort were chosen, we'd be no better off than just using that method in the first place.
- This is where a recursive “*leap of faith*” comes in:

WE CAN RECURSIVELY CALL OUR OWN MERGESORT METHOD

- **The basic idea (revised):**
 1. Split the array into two halves.
 2. mergeSort the left half.
 3. mergeSort the right half.
 4. Merge the two halves.

Recursive Merge Sort

```
public static void mergeSort(int[] a){
    if (a.length > 1){
        // 1. Split array into two halves.
        int[] left = Arrays.copyOfRange(a, 0, a.length / 2);
        int[] right = Arrays.copyOfRange(a, a.length / 2, a.length);

        // 2. and 3. Recursively call mergeSort to sort the two halves.
        mergeSort(left);
        mergeSort(right);

        // 4. Merge the two (sorted) halves.
        merge(a, left, right);
    }
}

public static void merge(int[] result, int[] left, int[] right){
    int iL = 0; // index into the left array (L)
    int iR = 0; // index into the right array (R)
    for(int i = 0; i < result.length; i++){
        if(iR >= right.length || (iL < left.length && left[iL] <= right[iR])){
            result[i] = left[iL]; // take from the left
            iL++;
        } else {
            result[i] = right[iR]; // take from the right
            iR++;
        }
    }
}
```

mergeSort splits the array that needs sorting into half repeatedly until the subarrays contain only **ONE** element each (the algorithm's *base case*).

This process of dividing a large problem (sorting the original array) into numerous *trivial* problems (sorting arrays of size = 1) allows merge sort to be a *much faster* sorting method than selection sort.

Program Complexity

- **Complexity** = A measure of the computing resources that are used by a piece of code, such as time, memory, or disk space.
- We generally mean *time complexity* when talking about the efficiency of a program.
- Time complexity can be evaluated ***empirically***, by measuring how long a program actually takes to run, but this is not very reliable and can vary greatly between machines.

Algorithm Analysis

- Mathematically approximate the performance of an algorithm
- For simplicity, assume the following operations require a ***fixed and equal amount*** of time to execute:
 - Variable declarations and assignments
 - Evaluating mathematical and logical expressions
 - Accessing or modifying an individual element of an array
 - Simple method calls (where the method does not perform a loop)

NOTE: initializing an array does not fall into this group of operations requiring a fixed amount of time – Java initializes each array element to ZERO and this takes longer for longer arrays.

Runtime of code that contains sequential loops

- The runtime of a loop is roughly equal to the runtime of its body x the number of iterations of the loop
- The runtime of **sequential** (not nested) loops is **additive**:
 - Runtime of loop 1 = M
 - Runtime of loop 2 = N
 - Runtime of both loops (one after the other) = $M + N$

Runtime of code that contains nested loops

- The runtime of nested loops is roughly equal to the runtime of the inner loop MULTIPLIED by the number of iterations of the outer loop:
 - # of iterations of outer loop = M
 - Running time inner loop = N
(N = running time of inner loop x # of iterations of inner loop)
 - Running time of the nested loop structure = $M \times N$
 - In general, we don't worry too much about the running time of the loop bodies (provided they do not contain additional loops), and the running time can be approximated by multiplying the number of iterations for each loop together
 - Thus, if both the inner and outer loop run for N iterations, the runtime would be roughly approximated as N^2

Complexity Classes

- **Complexity Class:** A set of algorithms that have a similar relationship between input data *size* and *resource consumption*.
- The complexity class of an algorithm is determined by looking at its most frequently executed line of code and determining how many times it is executed
- If the most frequently executed line gets executed $(2N^3 + 4N)$ times, the algorithm is in the “order N^3 ” complexity class, or **$O(N^3)$** in what is called **big-Oh notation** (*In big-Oh notation constants and lower terms are dropped*)

Common Complexity Classes

- Listed from slowest to fastest growth (lowest to highest complexity)
- The algorithms with the **slowest growth** are the ***fastest*** algorithms

Constant-time: $O(1)$
Logarithmic: $O(\log N)$
Linear: $O(N)$
Log-linear: $O(N \log N)$
Quadratic: $O(N^2)$
Cubic: $O(N^3)$
Exponential: (2^N)

Common Complexity Classes, cont.

Complexity Class	Description
<i>Constant-time, or $O(1)$</i>	Runtime does not depend on input size
<i>Logarithmic, or $O(\log N)$</i>	These algorithms typically divide a problem space in half repeatedly until the problem is solved (e.g. binary search)
<i>Linear, or $O(N)$</i>	Runtime is directly proportional to input size, N . Many algorithms that process each element in a data set are linear (e.g. count, sum, max, etc.)
<i>Log-linear, or $O(N \log N)$</i>	Typically perform a combination of logarithmic and linear operations, such as executing a logarithmic algorithm over every element of a data set of size N . Many efficient sorting algorithms, such as <i>merge sort</i> , are log-linear.

Common Complexity Classes, cont.

Complexity Class	Description
<i>Quadratic, or $O(N^2)$</i>	Runtime are proportional to the square of the input size, thus runtime quadruples when input size doubles . Selection sort is an example of a quadratic algorithm.
<i>Cubic, or $O(N^3)$</i>	Runtime are proportional to the cube of the input size. Code to multiply two $N \times N$ matrices would be an example of a cubic process.
<i>Exponential, or $O(2^N)$</i>	Runtimes are proportional to 2 raised to the power of the input size, N . This means that if the input size increased by JUST 1 , the running time of the algorithm doubles . Exponential algorithms are so slow that they should only be executed on very small datasets .