CSCI 136
Data Structures & Advanced Programming

Lecture 13

Spring 2020

Profs Bill & Dan

Administrative Details

- Lab 5: Sorting with Comparators
  - We give you the 2008 Williams student directory
  - We ask a series of questions that you can solve by sorting with Comparators and processing the sorted list
- Midterm: Wednesday March 18
  - Held in your scheduled Lab (same time and place)
  - Study guide and sample exam will be posted
  - Review session?

Last Time

- Induction practice
- Bubble sort
    - “Just buble sort, dude!”

Today's Outline

- `Comparable`: impose a sort order on objects
- `Comparator`: a class to implement sorting flexibly and modularly
- More "Simple" Sorting
  - ~~Bubble~~, Insertion, and Selection Sorts
  - General behavior
  - Big-O
  - Pros/cons

Objects?

- So far we’ve sorted integers using the notion of “<”, “>”, and “=”
- What about non-primitive types, like a `Patient` class?
  - We need a way to impose an ordering on arbitrary objects or else we can’t sort them
  - We need it to be flexible so we can reuse our sorting routines

java.lang.Comparable

- The Java language defines the `Comparable` Interface that sortable objects implement:

```
public interface Comparable<T>
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

java.lang.Comparable

- Objects that implement the `Comparable` interface must provide one method:

  `int compareTo(T o)`

  Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Comparable Interface

```
public interface Comparable<T> {

    //post: return < 0 if this smaller than other
    //      return 0 if this equal to other
    //      return > 0 if this greater than other
    int compareTo(T other);

}
```

- Any class that implements `Comparable` provides `compareTo()`

Notes on compareTo()

- The *magnitude* of the values returned by `compareTo()` are not important.
  - We only care if the return value is positive, negative, or 0!
    - Often we see -1, 0, 1, but it is up to the implementer

- `compareTo()` defines a "*natural ordering*" of Objects
  - There's nothing "*natural*" about it…

- We can use `compareTo()` to implement sorting algorithms on generic `List` data structures!

Using the Comparable Interface

```
public static void bubbleSort(int data[], int n) {
    int numSorted = 0;
    int index;
    while (numSorted < n) {
        for (index = 1; index < n-numSorted; index++) {
                if (data[index-1] > data[index])
                        swap(data, index-1, index);
        }
        // at least one more value is now in place
        numSorted++;
    }
}
```

Using the Comparable Interface

```
public static void bubbleSort(Comparable data[], int n) {
  int numSorted = 0;
  int index;
  while (numSorted < n) {
      for (index = 1; index < n-numSorted; index++) {
            if (data[index-1].compareTo(data[index]) > 0)
                  swap(data, index-1, index);
      }
      // at least one more value is now in place
      numSorted++;
  }
}
```

More Notes on compareTo()

- The `Comparable` interface (`Comparable<T>`) is part of the `java.lang` (not `structure5`) package.
- Other Java-provided structures can take advantage of objects that implement `Comparable`
    - Strings, or the `Arrays` class in `java.util`
- Note: Users of Comparable are urged to ensure that `compareTo()` and `equals()` are *consistent*. That is,
    - x.compareTo(y) == 0 exactly when x.equals(y) == true
- Note that `Comparable` limits user to a *single ordering*
- The syntax can get kind of dense
    - See BinSearchComparable.java : a generic binary search method
    - And even more cumbersome….

Comparators

- Limitations with Comparable interface?
  - `Comparable` permits `1` order between objects
    - What if `compareTo()` isn't the desired ordering?
  - What if `Comparable` isn't implemented?
- Solution: Comparators

Comparators (Ch 6.8)

- A comparator is an object that contains a method that is capable of comparing two objects
- Sorting methods can be written to apply a Comparator to two objects when a comparison is to be performed
- Different comparators can be applied to the same data to sort in different orders or on different keys

```
public interface Comparator <E> {
    // pre: a and b are valid objects
    // post: returns a value <, =, or > than 0 determined by
    // whether a is less than, equal to, or greater than b
    public int compare(E a, E b);
}
```

Example

Note that Patient does not implement Comparable or Comparator!

```
class Patient {
    protected String severity;
    protected String name;
    public Patient (String n, int a) { name = n; age = a; }
    public String getName() { return name; }
    public String getSeverity() { return severity; }
}
```

```
class SeverityComparator implements Comparator <Patient>{
    public int compare(Patient a, Patient b) {
       return a.getSeverity().compareTo(b.getSeverity());
    }
   // Note: No constructor; a “do-nothing” constructor is added by Java
}
```

```
public void <T> sort(T a[], Comparator<T> c) {
   …
   if (c.compare(a[i-1], a[i]) > 0) {…}
}
```

```
sort(patients, new SeverityComparator());
```

Comparable vs Comparator

- `Comparable` Interface for class `X`
  - Permits just one order between objects of class `X`
  - Class `X` must implement a `compareTo` method
  - Changing order requires rewriting `compareTo`
    - And then recompiling class `X` ☹
- `Comparator` Interface
  - Allows creation of “compator classes” for class `X`
  - Class `X` isn’t changed or recompiled
  - Multiple Comparators for `X` can be developed
    - Ex: Sort Strings by length (alphabetically for same-length)
    - Ex: Sort names by last name instead of first name

Sorting Preview: Bubble Sort

- Simple sorting algorithm that works by ascending through the list to be sorted, comparing two items at a time, and swapping them if they are in the wrong order
- Repeated until no swaps are needed
- Gets its name from the way larger elements "bubble" to the end of the list

Bubble Sort Example

5 1 3 2 9

- First Pass:
    - (**5** 1 3 2 9) → (1 **5** 3 2 9)
    - (1 **5** 3 2 9) → (1 3 **5** 2 9)
    - (1 3 **5** 2 9) → (1 3 2 **5** 9)
    - (1 3 2 **5** 9) → (1 3 2 5 **9**)
- Second Pass:
    - (**1** 3 2 5 9) → (**1** 3 2 5 9)
    - (1 **3** 2 5 9) → (1 2 **3** 5 9)
    - (1 2 **3** 5 9) → (1 2 3 **5** 9)
- Third Pass:
    - (**1** 2 3 5 9) -> (**1** 2 3 5 9)
    - (1 **2** 3 5 9) -> (1 **2** 3 5 9)
- Fourth Pass:
    - (**1** 2 3 5 9) -> (**1** 2 3 5 9)

http://www.youtube.com/watch?v=lyZQPjUT5B4

Sorting Analysis: Bubble Sort

- Worst-case time complexity?
  - O(n^2)
    - Each pass swaps all the way to the end
    - As described, doesn't recognize that list is sorted – keeps going
- Space complexity?
  - O(n)
    - It performs an *in-place* sort: no extra space is needed
- Stable?
  - Yes
    - The relative order of elements is preserved in the final array

Sorting Preview: Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Keep a sorted list in the low region of the array
- Keep the to-be-sorted part in the upper region
- Each round you “grow” the sorted region by swapping the first unsorted element backwards into its sorted location

Insertion Sort Example

- 5 7 0 3 4 2 6 1
- 5 7 0 3 4 2 6 1
- 0 5 7 3 4 2 6 1
- 0 3 5 7 4 2 6 1
- 0 3 4 5 7 2 6 1
- 0 2 3 4 5 7 6 1
- 0 2 3 4 5 6 7 1
- 0 1 2 3 4 5 6 7

Red: sorted region.
Each round, swap the first unsorted item back into sorted region

Sorting Analysis: Insertion Sort

- Worst-case time complexity?
  - $O(n^2)$
    - Each element may need to swap all the way back
    - Efficient on lists that are already substantially sorted. Actually has a best case of O(n)!
- Space complexity?
  - O(n)
    - It performs an *in-place* sort: no extra space is needed
- Stable?
  - Yes
    - The relative order of elements is preserved in the final array

Sorting Preview: Selection Sort

The algorithm works as follows:

- Find the maximum value in the list
- Swap it with the value in the last position (since the last position is the place where the maximum element goes)
- Repeat the steps above for the unsorted prefix of the list

Sorting Preview: Selection Sort

11	3	$\boxed{27}$	5	16	Swap 27 with 16
11	3	$\boxed{16}$	5	$\underline{27}$	Swap 16 with 5
$\boxed{11}$	3	5	$\underline{16}$	$\underline{27}$	Swap 11 with 5
$\boxed{5}$	3	$\underline{11}$	$\underline{16}$	$\underline{27}$	Swap 5 with 3
3	$\underline{5}$	$\underline{11}$	$\underline{16}$	$\underline{27}$	Done!

Sorting Analysis: Selection Sort

- Similar to insertion sort, but performs worse than insertion sort in general

- Worst-place time complexity?
  - O(n^2)
    - Need to scan the list for the largest element every round, but only "swaps" once per round
    - As described, doesn't recognize that list is sorted – keeps going
- Space complexity?
  - O(n)
    - It performs an *in-place* sort: no extra space is needed

“Basic” Sorting Algorithm Recap

- BubbleSort
  - Swaps consecutive elements of a[0..k] until largest element is at a[k]; Decrements k and repeats
- InsertionSort
  - Assumes a[0..k] is sorted and swaps a[k+1] backwards across a[0..k] until a[0..k+1] is sorted
  - Increments k and repeats
- SelectionSort
  - Finds largest item in a[0..k] and swaps it with a[k]
  - Decrements k and repeats

Basic Sorting Algorithms

(All have worst-case $O(n^2)$ runtime)

- BubbleSort
  - Always performs cn^2 comparisons and might need to perform cn^2 swaps
- InsertionSort
  - Might perform cn^2 comparisons and cn^2 swaps, but in best case cn comparisons and 0 swaps
- SelectionSort
  - Always performs cn^2 comparisons but only $O(n)$ swaps

Swap!

- The “Basic” sorts all use a utility method: `swap`. How would you implement `swap`?

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

Comparators

- Limitations with Comparable interface?
  - `Comparable` permits `1` order between objects
  - What if `compareTo()` isn't the desired ordering?
  - What if `Comparable` isn't implemented?
- Solution: Comparators

Comparators (Ch 6.8)

- A comparator is an object that contains a method that is capable of comparing two objects
- Sorting methods can be written to apply a Comparator to two objects when a comparison is to be performed
- Different comparators can be applied to the same data to sort in different orders or on different keys

```
public interface Comparator <E> {
     // pre: a and b are valid objects
     // post: returns a value <, =, or > than 0 determined by
     // whether a is less than, equal to, or greater than b
     public int compare(E a, E b);
}
```

Example

Note that Patient does not implement Comparable or Comparator!

```
class Patient {
    protected int age;
    protected String name;
    public Patient (String n, int a) { name = n; age = a; }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```

```
class NameComparator implements Comparator <Patient>{
    public int compare(Patient a, Patient b) {
       return a.getName().compareTo(b.getName());
    }
   // Note: No constructor; a “do-nothing” constructor is added by Java
}
```

```
public void <T> sort(T a[], Comparator<T> c) {
   …
   if (c.compare(a[i], a[max]) > 0) {…}
}
```

```
sort(patients, new NameComparator());
```

Comparable vs Comparator

- `Comparable` Interface for class `X`
  - Permits just one order between objects of class `X`
  - Class `X` must implement a `compareTo` method
  - Changing order requires rewriting `compareTo`
    - And then recompiling class `X`
- `Comparator` Interface
  - Allows creation of "compator classes" for class `X`
  - Class `X` isn't changed or recompiled
  - Multiple Comparators for `X` can be developed
    - Ex: Sort Strings by length (alphabetically for same-length)
    - Ex: Sort names by last name instead of first name

Selection Sort with Comparator

```
public static <E> int findPosOfMax(E[] a, int last,
                                   Comparator<E> c) {
      int maxPos = 0       // A wild guess
      for(int i = 1; i <= last; i++)
            if (c.compare(a[maxPos], a[i]) < 0)
                  maxPos = i;
      return maxPos;
}
public static <E> void selectionSort(E[] a, Comparator<E> c) {
      for(int i = a.length - 1; i>0; i--) {
          int big= findPosOfMin(a,i,c);
          swap(a, i, big);
      }
}
```

- The same array can be sorted in multiple ways by passing different `Comparator<E>` values to the sort method;

Merge Sort

- A divide and conquer algorithm
- Merge sort works as follows:
  - Base case:
    - If the list is of length 0 or 1, then it is already sorted. Return the sorted list.
  - Divide the unsorted list into two sublists of about half the size of original list.
  - Recursive call:
    - Sort each sublist by re-applying merge sort.
  - Merge the two sublists back into one sorted list.

Merge Sort

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

Transylvanian Merge Sort Folk Dance

Merge Sort

- How would we implement it?
- Pseudocode:

```
//recursively mergesorts A[from..To] “in place”
void recMergeSortHelper(A[], int from, int to)
  if ( from < to )
     // find midpoint
     mid = (from + to)/2
     //sort each half
     recMergeSortHelper(A, from, mid)
     recMergeSortHelper(A, mid+1, to)
     // merge sorted lists
     merge(A, from, to)
```

But `merge` hides a number of important details….

Merge Sort

- How would we implement it?
  - Review `MergeSort.java`
  - Note carefully how temp array is used to reduce copying
  - Make sure the data is in the correct array!
- Time Complexity?
  - Takes at most `2k` comparisons to merge two lists of size `k`
  - Number of splits/merges for list of size `n` is `log n`
  - Claim: At most time `O(n log n)`…We'll see soon...
- Space Complexity?
  - `O(n)`?
  - Need an extra array, so really `O(2n)`!
    - But `O(2n) = O(n)`

Merge Sort = O(n log n)

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

log n

log n

merge takes at most n comparisons per line

Merge Sort

- Unlike Bubble, Insertion, and Selection sort, Merge sort is a divide and conquer algorithm
  - Bubble, Insertion, Selection sort: `O(n`^{2}`)`
  - Merge sort: `O(n log n)`
- Are there any problems or limitations with Merge sort?
- Why would we ever use any other algorithm for sorting?

Problems with Merge Sort

- Need extra temporary array
    - If data set is large, this could be a problem
- Waste time copying values back and forth between original array and temporary array
- Can we avoid this?

Quick Sort

- Quick sort is designed to behave much like Merge sort, without requiring extra storage space

Merge Sort	Quick Sort
Divide list in half	Partition* list into 2 parts
Sort halves	Sort parts
Merge halves	Join* sorted parts

Recall Merge Sort

```
private static void mergeSortRecursive(Comparable data[],
                    Comparable temp[], int low, int high) {
   int n = high-low+1;
   int middle = low + n/2;
   int i;

   if (n < 2) return;
   // move lower half of data into temporary storage
   for (i = low; i < middle; i++) {
       temp[i] = data[i];
   }
   // sort lower half of array
   mergeSortRecursive(temp,data,low,middle-1);
   // sort upper half of array
   mergeSortRecursive(data,temp,middle,high);
   // merge halves together
   merge(data,temp,low,middle,high);
}
```

Quick Sort

```
// pre: low <= high
// post: data[low..high] in ascending order
public void quickSortRecursive(Comparable data[],
                    int low, int high) {
        int pivot;
        /* base case: low and high coincide */
        if (low >= high) return;

        /* step 1: split using pivot */
        pivot = partition(data, low, high);
        /* step 2: sort small */
        quickSortRecursive(data, low, pivot-1);
        /* step 3: sort large */
        quickSortRecursive(data, pivot+1, high);
}
```

Partition

1. Put first element (pivot) into sorted position
2. All to the left of “pivot” are smaller and all to the right are larger
3. Return index of “pivot”

Partition by Hungarian Folk Dance

Partition

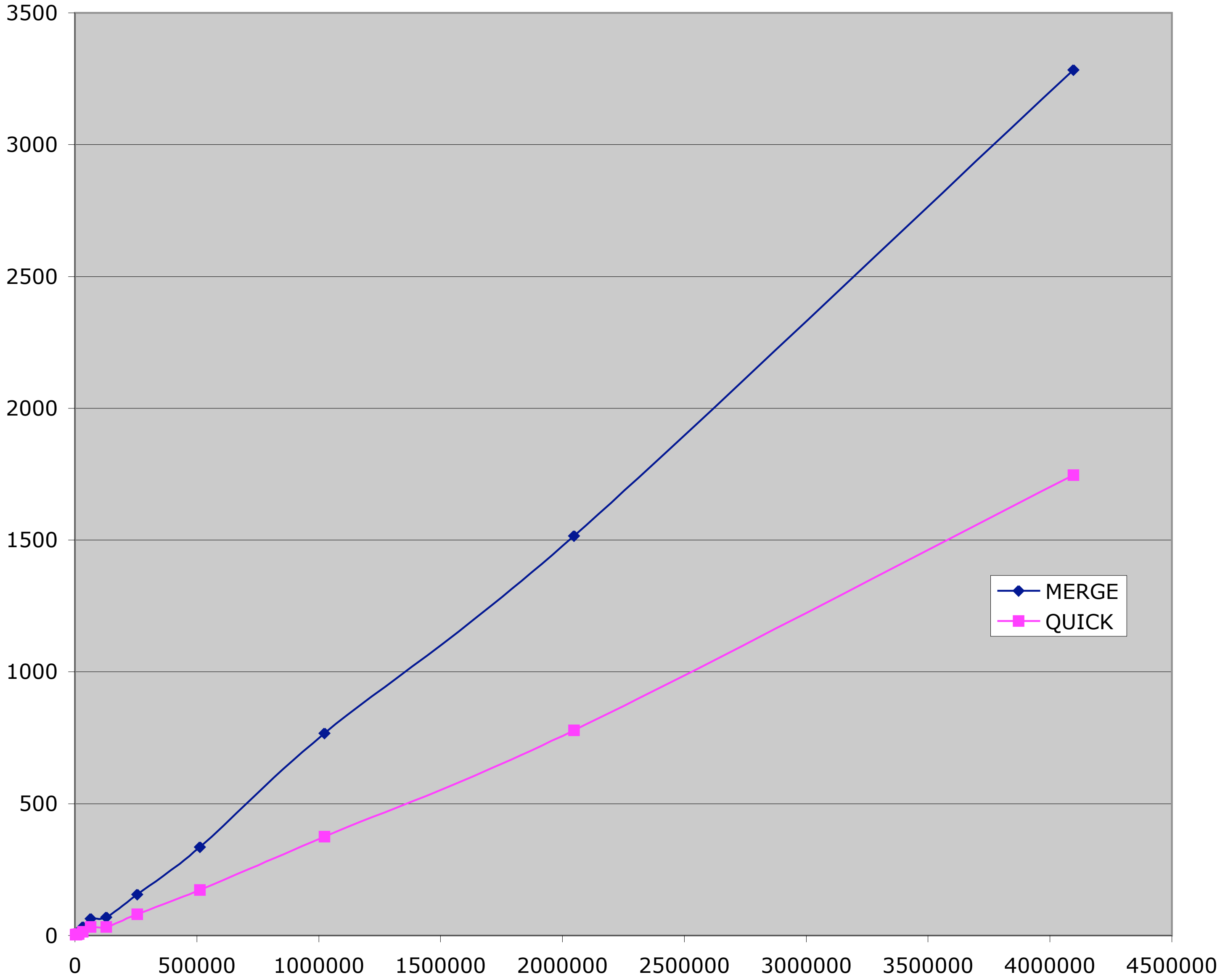
```
int partition(int data[], int left, int right) {
  while (true) {
    while (left < right && data[left] < data[right])
      right--;
    if (left < right) {
      swap(data, left++, right);
    } else {
      return left;
    }

    while (left < right && data[left] < data[right])
      left++;
    if (left < right) {
      swap(data, left, right--);
    } else {
      return right;
    }
  }
}
```

Complexity

- Time:
  - Partition is O(n)
  - If partition breaks list exactly in half, same as merge sort, so O(n log n)
  - If data is already sorted, partition splits list into groups of 1 and n-1, so O(n^2)
- Space:
  - O(n) (so is MergSort)
    - In fact, it's n + c compared to 2n + c for MergeSort

Merge vs. Quick



Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot (heuristic!)
- Combine selection sort with quick sort
  - For small n, selection sort is faster
  - Switch to selection sort when elements is <= 7
  - Switch to selection/insertion sort when the list is almost sorted (partitions are very unbalanced)
    - Heuristic!

Sorting Wrapup

	Time	Space
Bubble	Worst: $O(n^2)$ Best: $O(n)$ - if "optimiazed"	$O(n)$: $n + c$
Insertion	Worst: $O(n^2)$ Best: $O(n)$	$O(n)$: $n + c$
Selection	Worst = Best: $O(n^2)$	$O(n)$: $n + c$
Merge	Worst = Best:: $O(n \log n)$	$O(n)$: $2n + c$
Quick	Average = Best: $O(n \log n)$ Worst: $O(n^2)$	$O(n)$: $n + c$

More Skill-Testing (Try these at home)

Given the following list of integers:

9 5 6 1 10 15 2 4

1) Sort the list using Bubble sort. Show your work!
2) Sort the list using Insertion sort. Show your work!
3) Sort the list using Merge sort. Show your work!
4) Verify the best and worst case time and space complexity for each of these sorting algorithms as well as for selection sort.