Administrative Details

• Lab 4 Wednesday: Sorting!
  • The lab has been posted on the Labs page
  • You may again work with a partner
    • Needn’t be same partner as Lab 3
    • Fill out the Google Form!
  • Produce a design before lab
    • Both members of pair should produce their own
Last Time

- Strong Induction
- The Comparable Interface
- Basic Sorting
  - Bubble, Insertion, Selection Sorts
  - Including time and space analysis
This Time

- More Comparable Examples
- Better Sorting Methods
  - MergeSort
  - QuickSort
- More Flexible Comparing: Comparator Interface
Java provides the `Comparable` interface, which specifies a method `compareTo()`

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

```java
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);
}
```
We could have written

```java
public class CardRankSuit implements Comparable<CardRankSuit> {
    public int compareTo(CardRankSuit other) {
        if (this.getSuit() != other.getSuit())
            return this.getSuit().compareTo(other.Suit());
        else
            return this.getRank().compareTo(other.getRank());
    }

    // rest of code for the class....
}
```
compareTo in Card Example

Notes

• enum types automatically implement Comparable
• The magnitude of the values returned by compareTo are not important. We only care if value is positive, negative, or 0!
• compareTo defines a “natural ordering” of Objects
  • There’s nothing “natural” about it….
• We use the BubbleSort algorithm to sort the cards in CardDeck.java
Comparable & 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
  • See the Arrays class in java.util
  • Example JavaArraysBinSearch
• 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....
ComparableAssociation

- Suppose we want an *ordered* Dictionary, so that we can use binary search instead of linear
- Structure5 provides a ComparableAssociation class that implements Comparable.
- The class declaration for ComparableAssociation is
  
  ```java
  public class ComparableAssociation<K extends Comparable<K>, V>
    Extends Association<K, V> implements Comparable<
    ComparableAssociation<K, V>>
  ```
  (Yikes!)
- Example: Since Integer implements Comparable, we can write
  - `ComparableAssociation<Integer, String> myAssoc =
    new ComparableAssociation( new Integer(567), “Bob”);
- We could then use Arrays.sort on an array of these
Faster Sorting: Merge Sort

• A divide and conquer algorithm
• Typically used on arrays
• Merge sort works as follows:
  • If the array is of length 0 or 1, then it is already sorted.
  • Divide the unsorted array into two arrays of about half the size of original.
  • Sort smaller arrays recursively by re-applying merge sort.
  • Merge the two smaller arrays back into one sorted array.
• Time Complexity?
  • Spoiler Alert! We’ll see that it’s $O(n \log n)$
• Space Complexity?
  • $O(n)$
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
Merge Sort

• How would we implement it?
• First pass…

// recursively mergesorts A[from .. To] “in place”
void recMergeSortHelper(A[], int from, int to)
    if (from ≤ to)
        mid = (from + to)/2
        recMergeSortHelper(A, from, mid)
        recMergeSortHelper(A, mid+1, to)
        merge(A, from, to)

But merge hides a number of important details….
Merge Sort = \( O(n \log n) \)

- \([8 \ 14 \ 29 \ 1 \ 17 \ 39 \ 16 \ 9]\)  
  - 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

merge takes at most \( n \) comparisons per line
Time Complexity Proof

• Prove for \( n = 2^k \) (true for other \( n \) but harder)
• That is, MergeSort for \( n \) performs at most
  • \( n \cdot \log(n) = 2^k \cdot k \) comparisons of elements
• Base cases \( k \leq 1 \): 0 comparisons: \( 0 < 1 \cdot 2^1 \) ✓
• Induction Step: Suppose true for all integers smaller than \( k \). Let \( T(k) \) be # of comparisons for \( 2^k \) elements. Then
  • \( T(k) \leq 2^k + 2T(k-1) \leq 2^k + 2(k-1)2^{k-1} \leq k \cdot 2^k \) ✓
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.

<table>
<thead>
<tr>
<th>Merge Sort</th>
<th>Quick Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide list in half</td>
<td>Partition* list into 2 parts</td>
</tr>
<tr>
<td>Sort halves</td>
<td>Sort parts</td>
</tr>
<tr>
<td>Merge halves</td>
<td>Join* sorted parts</td>
</tr>
</tbody>
</table>
public void quickSortRecursive(Comparable data[],
        int low, int high) {
    // pre: low <= high
    // post: data[low..high] in ascending order
    int pivot;
    if (low >= high) return;

    /* 1 - place pivot */
    pivot = partition(data, low, high);
    /* 2 - sort small */
    quickSortRecursive(data, low, pivot-1);
    /* 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”
```c
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 (Average Time)
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

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>Worst: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Best: $O(n)$ - if “optimized”</td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Worst: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Best: $O(n)$</td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>Worst = Best: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td>Merge</td>
<td>Worst = Best: $O(n \log n)$</td>
<td>$O(n) : 2n + c$</td>
</tr>
<tr>
<td>Quick</td>
<td>Average = Best: $O(n \log n)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Worst: $O(n^2)$</td>
<td></td>
</tr>
</tbody>
</table>
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.
Comparators

- Limitations with Comparable interface
  - Only permits one order between objects
  - What if it isn’t the desired ordering?
  - What if it 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

```java
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

```java
class Patient {
    protected int age;
    protected String name;
    public Patient (String s, int a) {name = s; 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 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 recompiling class X

- Comparator Interface
  - Allows creation of “Comparator classes” for class X
  - Class X isn’t changed or recompiled
  - Multiple Comparators for X can be developed
    - Sort Strings by length (alphabetically for equal-length)
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;