CSCI 136
Data Structures &
Advanced Programming

Lecture 9

Fall 2019

Instructors: B&S

Administrative Details

- Remember: First Problem Set is online
  - Due at beginning of class on Friday
- Lab 3 Today!
  - You *may* work with a partner
  - Come to lab with a plan!
  - Answer questions before lab
- Lab 1 has been returned….

Last Time

- Mathematical Induction
  - For algorithm run-time and correctness
- More About Recursion
  - Recursion on arrays; helper methods

Today's Outline

- Final Tips on Induction
- Basic Sorting
  - Bubble, Insertion, Selection Sorts
  - Including proofs of correctness
- The Comparable Interface
  - An Extended Example: A Playing Card Type
  - Making Search Generic

Notes on Induction

- Whenver induction is needed, strong induction can be used
- The numbering of the propositions doesn't need to start at 0
- The number of base cases depends on the problem at hand
  - Enough are needed to guarantee that the smallest non-base case can be proven using only the base cases

Bubble Sort

- 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
http://www.visualgo.net/sorting

Sorting Intro: Bubble Sort

- Simple sorting algorithm that works by repeatedly stepping 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
- Time complexity?
  - $O(n^2)$: Might perform $O(n^2)$ compares *and* $O(n^2)$ swaps
- Space complexity?
  - $O(n)$ total (very little additional space is required)

Sorting Intro: Insertion Sort

- 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

http://www.visualgo.net/sorting

Sorting Intro : Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Less efficient on large lists than more advanced algorithms
- Advantages:
  - Simple to implement and efficient on small lists
  - Efficient on data sets which are already mostly sorted
- Time complexity : Worst Case
  - $O(n^2)$: Could perform $O(n^2)$ compares and $O(n^2)$ moves
- Space complexity
  - $O(n)$

Sorting Intro : Selection Sort

http://www.visualgo.net/sorting

(demo is "min" version)

- 11 3 27 5 16
- 11 3 16 5 27
- 11 3 5 16 27
- 5 3 11 16 27
- 3 5 11 16 27

- Time Complexity:
  - $O(n^2)$: Might perform $O(n^2)$ compares; only $O(n)$ swaps
- Space Complexity:
  - $O(n)$

Sorting Intro : Selection Sort

- Similar to insertion sort
- Noted for its simplicity and performance advantages when compared to complicated algorithms
- The algorithm works as follows:
  - Find the maximum value in the list
  - Swap it with the value in the last position
  - Repeat the steps above for remainder of the list (ending at the second to last position)

Some Skill Testing!

Selection sort uses two utility methods

Uses a swap method

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

And a max-finding method

```
// Find position of largest value in A[0 .. last]
public static int findPosOfMax(int[] A, int last) {
        int maxPos = 0;     // A wild guess
        for(int i = 1; i <= last; i++)
                if (A[maxPos] < A[i]) maxPos= i;
        return maxPos;
}
```

Some Skill Testing!

An Iterative Selection Sort

```
public static void selectionSort(int[] A) {
       for(int i = A.length - 1; i>0; i--)
           int big= findPosOfMax(A,i);
           swap(A, i, big);
       }
}
```

A Recursive Selection Sort (just the helper method)

```
public static void recSSHelper(int[] A, int last) {
       if(last == 0) return; // base case

       int big= findPosOfMax(A, last);
       swap(A,big,last);
       recSSHelper(A, last-1);
}
```

Some Skill Testing!

- Prove: recSSHelper (A, last) sorts elements A[0]…A[last].
  - Assume that maxLocation(A, last) is correct
- Proof:
  - Base case: last = 0.
  - Induction Hypothesis:
    - For k<last, recSSHelper sorts A[0]…A[k].
  - Prove for last:
    - Note: Using Second Principle of Induction (Strong)

Some Skill Testing!

- After call to findPosOfMax(A, last):
  - 'big' is location of largest A[0..last]
- That value is swapped with A[last]:
  - Rest of elements are A[0]..A[last−1].
- Since last − 1< last, then by induction
  - recSSHelper(A, last−1) sorts A[0]..A[last−1].
- Thus A[0]..A[last−1] are in increasing order
  - *and* A[last−1] ≤ A[last].
- So, A[0]···A[last] are sorted.

Making Sorting Generic

- We need *comparable* items
- Unlike with equality testing, the Object class doesn't define a "compare()" method 😟
- We want a uniform way of saying objects can be compared, so we can write generic versions of methods like binary search
- Use an interface!
- Two approaches
  - Comparable interface
  - Comparator interface

Java Interfaces : Motivating Example

- Idea: Implement a class that describes a single playing card (e.g., "Queen of Diamonds")
- Start simple: a single class – BasicCard
- Think about alternative implementations
- Use an *interface* to allow implementation independent coding
- Let's look at BasicCard

Aside : Enum Types are Class Types

```
enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN,
      EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE;
}
```

Notes

- Creates an ordered sequence of named constants
- Can find position of an enum value in sequence
  - `int i = r.ordinal(); // r is of type Rank`
- Can get an array of all values in the enum
  - `Rank[] allRanks = Rank.values();`
- Can use in **for** loops
  - `for (Rank r : Rank.values() ) { ... }`
- Can have its own instance variables and methods

Implementing a Card Object

- Think before we code!
- Many ways to implement a card
  - An index from 0 to 51; a rank and a suit, ...
- Start general.
  - Build an *interface* that advertises all public features of a card
  - Not an implementation (define methods, but don't include code)
- Then get specific.
  - Build specific implementation of a card using our general card interface

Start General: Card: An Interface

- What data do we have to represent?
  - Properties of cards
  - How can we represent these properties?
    - There are often multiple options—name some!
- What methods do we need?
  - Capabilities of cards
  - Do we need *accessor* and/or *mutator* methods?

A Card Interface

```
public interface Card {

    // Methods - must be public
    public Suit getSuit();
    public Rank getRank();
}
```

Notes

- Don’t allow card to change its value
  - Only need accessor methods
- Support `enums` for rank and suit

Get Specific: Card Implementations

- Now suppose we want to build a specific card object
- We want to use the properties/capabilities defined in our interface
  - That is, we want to *implement* the interface

```
public class CardRankSuit implements Card {
    . . .
}
```

The Enums for Cards

```
public enum Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES; // the values

public String toString() {
        switch (this) {
        case CLUBS : return "clubs";
        case DIAMONDS : return "diamonds";
        case HEARTS : return "hearts";
        case SPADES : return "spades";
        }
        return "Bad suit!";
    }
}
```

A similar declaration is defined for Rank

A First Card Implementation

```
public class CardRankSuit implements Card {
// instance variables
      protected Suit suit;
      protected Rank rank;
// Constructors
      public CardRankSuit( Rank r, Suit s)
            {suit = s; rank = r;}
// returns suit of card
      public Suit getSuit() { return suit;}
// returns rank of card
      public Rank getRank() { return rank;}
// create String representation of card
 public String toString()
           {return getRank() + " of " + getSuit();}
}
```

A Second Card Implementation

```
public class Card52 implements Card {
// instance variables
protected int code; // 0 <= code < 52;
// rank is code % 13 and suit is code / 13
// Constructors
public CardRankSuit( int index )
      {code = index;}
// returns suit of card
      public Suit getSuit() {// see sample code}
// returns rank of card
      public Rank getRank() {// see sample code}
// create String representation of card
 public String toString()
           {return getRank() + " of " + getSuit();}
}
```

Improvements to Card52

Add back a constructor with Rank/Suit parameters

```
public class Card52v2 implements Card {
...
public Card52v2( Rank theRank, Suit theSuit) {
  code = theSuit.ordinal() * 13 + theRank.ordinal();
}
```

Replace switch statements in “get” methods...

```
public Suit getSuit() {
        return Suit.value( code / 13 );}
public Rank getRank() {
        return Rank.value( code % 13 );}
```

...by adding value() method to each enum

```
 public static Rank value(int ordVal) {
        return vals[ordVal];}
```

Interfaces: Worth Noting

- Interface methods **are always** public
  - Java does not allow non-public methods in interfaces
- Interface instance variables are always **static final**
  - static variables are shared across instances
  - final variables are constants: they can't change value
- Most classes contain constructors; interfaces do not!
- Can *declare* interface objects (just like class objects) but cannot instantiate ("new") them
- Typically there is no executable code in an Interface
  - Although it is possible to include code in certain situations

Searching & Sorting
The Comparable Interface

- Java provides an interface for comparisons between objects
  - Provides a replacement for “<“ and “>” in recBinarySearch
- Java provides the *Comparable* interface, which specifies a method *compareTo()*
  - Any class that implements Comparable must provide compareTo()

```
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);
}
```

Comparable Interface

- Many Java-provided classes implement Comparable
  - String (alphabetical order)
  - Wrapper classes: Integer, Character, Boolean
  - All Enum classes
- We can write methods that work on any type that implements Comparable
  - Let's See some examples
    - RecBinSearch.java
    - BinSearchComparable.java

compareTo in Card Example

We could write

```
public class CardRankSuit implements
      Comparable<CardRankSuit> {

   public int compareTo(CardRankSuit other) {
      if (this.getSuit() != other.getSuit())
         return getSuit().compareTo(other.Suit());
      else
         return getRank().compareTo(other.getRank());
   }
// rest of code for the class....
}
```

compareTo in Card Example

We actually wrote (in Card.java)

```
public interface Card extends Comparable<Card> {
   public int compareTo(Card other);
   // remainder of interface code
}
```

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

…wait for it...

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