CSCI 136:
Data Structures and
Advanced Programming

Lecture 14
Sorting, part 2

Instructor: Dan Barowy
Williams

Topics
• Association
• Bubble sort complexity
• How do we sort data of any type?
• Other sorts

Your to-dos
1. Read before Fri: Bailey, Ch 6.5-6.6.
2. Quiz 5, due Saturday by 6pm.
3. Lab 5, due Tuesday 10/18 by 10pm.

Miiiiiiidterrerrrrrm!
Announcements

1. Midterm: in lab two weeks from now:
   - Wed, October 26 and
   - Thu, October 27
2. Midterm review: Mon, October 24 in class.
3. No class: Fri, October 28.

Simple data structure for Lab 5...

Association

```java
public class Association<K, V> extends java.lang.Object
implements java.util.Map.Entry<K, V>
```

A class implementing a key-value pair. This class associates an immutable key with a mutable value. Used in many other structures.

Example Usage:

```java
public static void main(String[] args) {
    // store the number of classes taken by the student in an array of associations
    Association[] classesTaken = new Association[5];
    classesTaken[0] = new Association("Andrew", new Integer(3));
    classesTaken[1] = new Association("Barbara", new Integer(3));
    classesTaken[2] = new Association("Bill", new Integer(3));
    classesTaken[3] = new Association("Dan", new Integer(3));
    classesTaken[4] = new Association("Tom", new Integer(3));

    // print out each item in the array
    for (int i = 0; i < classesTaken.length; i++) {
        System.out.println("This Student has taken "+ classesTaken[i].getValue() +
            " classes from "+ classesTaken[i].getKey() + ",/";)
    }
}
```

You will need to use Association in Lab 5.

Sorting
Recall: bubble sort

```java
public static void bubbleSort(int data[], int n)
    // pre: 0 <= n <= data.length
    // post: values in data[0..n-1] in ascending order
{
    int numSorted = 0; // number of values in order
    int index; // general index
    while (numSorted < n)
    {
        // bubble a large element to higher array index
        for (index = 1; index < n-numSorted; index++)
        {
            if (data[index-1] > data[index])
                swap(data,index-1,index);
        }
        // at least one more value in place
        numSorted++;
    }
}
```

Bubble sort complexity

Bubble sort is an $O(n^2)$ sorting algorithm in the worst case. The naive algorithm is also $O(n^2)$ in the best case. With a small modification, bubble sort is $O(n)$ in the best case (i.e., where the array is already sorted).

Bubble sort’s performance is bad enough that there are few practical uses for it (other than for teaching!).

What if…

… you wanted to sort data that isn’t just a bunch of ints?

What’s problematic with our bubble sort implementation?
Where is the problem?

We frequently have to sort data that is more complex than simple numbers.

For example, suppose we need to sort objects, like a `People[]`.

How do we define an order so that we can easily sort this?

`compare` to the rescue.

Comparators

The `Comparator interface` defines the method `compare` that lets us compare two elements of the same type.

```java
public int compare(T o1, T o2)
```

Returns any `int < 0` when `o1` is “less than” `o2`.

Returns any `int > 0` when `o2` is “less than” `o1`.

Returns `0` otherwise.
Let’s modify this algorithm

```java
public static void bubbleSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0..n-1] in ascending order
{
    int numSorted = 0; // number of values in order
    int index; // general index
    while (numSorted < n)
    {
        // bubble a large element to higher array index
        for (index = 1; index < n-numSorted; index++)
        {
            if (data[index-1] > data[index])
                swap(data,index-1,index);
        }
        // at least one more value in place
        numSorted++;
    }
}
```

Better comparison sorts
**Insertion sort**

Insertion sort is a **sorting algorithm** in which the next element is “inserted” into a sorted array during each step. Insertion sort makes \( n-1 \) passes through the sorted data, performing pairwise comparisons of elements using \(<\).

Insertion sort maintains the **invariant** that the leftmost \( 	ext{numSorted} \) elements are sorted.

I.e., insertion sort builds a sorted order to the left.

**Insertion sort complexity**

Insertion sort is an \( \mathcal{O}(n^2) \) sorting algorithm in the **worst case**. Insertion sort is \( \mathcal{O}(n) \) in the best case.

```java
public static void insertionSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0...n-1] are in ascending order
{
    int numSorted = 1;  // number of values in place
    int index;          // general index
    while (numSorted < n)
    {
        // take the first unsorted value
        int temp = data[numSorted];
        // ...and insert it among the sorted:
        for (index = numSorted; index > 0; index--)
        {
            if (temp < data[index-1])
            {
                data[index] = data[index-1];
                break;
            }
        }
        // reinsert value
        data[index] = temp;
        numSorted++;
    }
}
```
On your own…

Read about selection sort from the book.

Recap & Next Class

Today:

• Association
• Sort complexity
• Comparators

Next class:

• Very fast comparison sorts