Administrative Details

- Lab 4 Today!
  - Try to answer questions before lab
- Mountain Day Madness!
  - If This Friday is Mountain Day
    - Lab 5 will go on-line this weekend
    - Problem Set 2---coming this Friday---will also go on-line this weekend (due next Friday at start of class)
    - And---OMG---we won’t see you again until next Wednesday!!!
Last Time

• More about Mathematical Induction
  • For algorithm run-time and correctness
• More About Recursion
  • Recursion on arrays; helper methods
  • Recursion on Chains
• Strong Induction
• Linear and Binary Searching review
Today’s Outline

• The Comparable Interface
• Basic Sorting
  • Bubble, Insertion, Selection Sorts
  • Including proofs of correctness
And, if time permits…
• Comparator interfaces for flexible sorting
• More Efficient Sorting Algorithms
  • MergeSort, QuickSort
public class BinSearch {

    public static int binarySearch(int a[], int value) {
        return recBinarySearch(a, value, 0, a.length-1); }

    protected static int recBinarySearch(int a[], int value, int low, int high) {

        if (low > high) return -1;
        else {
            int mid = (low + high) / 2; //find midpoint
            if (a[mid] == value) return mid; //first comparison
            else if (a[mid] < value) //search upper half
                return recBinarySearch(a, value, mid + 1, high);
            else //search lower half
                return recBinarySearch(a, value, low, mid - 1);
        }
    }
}
Recall: Binary Search

• Why does it work?
  • Because items can be ordered (they are comparable)
  • So they can be sorted then searched based on ordering

• Why is it fast?
  • Cut `search space in half with each comparison!

• Requires items to be comparable

• If items are not comparable, we typically need to do a linear search
Linear Search

• Complexity analysis of linear search:
  • Best case: $O(1)$
  • Worst case: $O(n)$
  • Average case: $O(n)$
• Recall
  • Assume all locations equally likely
  • The average number of comparisons is $(1 + 2 + 3 + \ldots + n)/n = (n+1)/2$, so $O(n)$
• Here’s a generic linear search method
public class LinearSearchGeneric {
    // post: returns index of value in a, or -1 if not found
    // Note the <E> between static and int: a generic method!
    public static <E> int linearSearch(E a[], E value) {
        for (int i = 0; i < a.length; i++) {
            if (a[i].equals(value)) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String args[]) {
        // search a String array
        System.out.println(linearSearch(args, "cow"));
        // search an Integer array
        Integer odds[] = new Integer[] { 1,3,5,7,9 };  
        System.out.println(linearSearch(odds, 7));
    }
}
Linear vs. Binary Search

- Clearly binary is preferable
- But it requires ordered (i.e., sorted) data.
  - 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! (We’ll see two approaches)
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, 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);
}
```
compareTo in Card Example

We could have written

```java
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)

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

And in CardAbstract.java, we added

```java
public int compareTo(Card other) {
    if (this.getSuit() != other.getSuit())
        return getSuit().compareTo(other.Suit());
    else
        return getRank().compareTo(other.getRank());
}
```
As a result, all of our implementations of the Card interface have comparable card types!
compareTo in Card Example

Notes

• Enum types implement Comparable and define compareTo
• 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....
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>>
```

Example: Since Integer implements Comparable, we can write

```java
ComparableAssociation<Integer, String> myAssoc =
    new ComparableAssociation<>( new Integer(567), "Bob");
```

We could then use Arrays.sort on an array of these
Given an array $a[]$ of integers and a target integer $T$, is there a subset of the integers in the array that sum to $T$?

Example $a[] = 10, 7, 12, 3, 5, 11, 8, 9, 1, 15$:
- $T = 31$? Yes: $10 + 7 + 5 + 9$
- $T = 79$? No. [Why?]

How could we solve this problem?
- Hint: Either we use $a[0]$ or we don’t….  
- Need: `canMakeSumHelper(int set[], int target, int index)`

How could we prove our method was correct?
Complexity Analysis of Subset Sum

• The Subset Sum algorithm we wrote is slow.
• How slow?
• Let $s_n$ be the minimum number of steps the algorithm might take on an array of size $n$.
  • $s_n \geq 1 + s_{n-1} + s_{n-1} > 2s_{n-1}$
  • $s_1 = 1$
• Claim: $s_n \geq 2^{n-1}$—an exponential lower bound
  • Proof: Induction. [Easy: try it for homework]
• Can also prove an upper bound of $O(2^n)$
Bubble Sort

• First Pass:
  • \((5\ 1\ 3\ 2\ 9) \Rightarrow (1\ 5\ 3\ 2\ 9)\)
  • \((1\ 5\ 3\ 2\ 9) \Rightarrow (1\ 3\ 5\ 2\ 9)\)
  • \((1\ 3\ 5\ 2\ 9) \Rightarrow (1\ 3\ 2\ 5\ 9)\)
  • \((1\ 3\ 2\ 5\ 9) \Rightarrow (1\ 3\ 2\ 5\ 9)\)

• Second Pass:
  • \((1\ 3\ 2\ 5\ 9) \Rightarrow (1\ 3\ 2\ 5\ 9)\)
  • \((1\ 3\ 2\ 5\ 9) \Rightarrow (1\ 3\ 2\ 5\ 9)\)
  • \((1\ 2\ 3\ 5\ 9) \Rightarrow (1\ 2\ 3\ 5\ 9)\)

• Third Pass:
  • \((1\ 2\ 3\ 5\ 9) \Rightarrow (1\ 2\ 3\ 5\ 9)\)
  • \((1\ 2\ 3\ 5\ 9) \Rightarrow (1\ 2\ 3\ 5\ 9)\)

• Fourth Pass:
  • \((1\ 2\ 3\ 5\ 9) \Rightarrow (1\ 2\ 3\ 5\ 9)\)

http://www.youtube.com/watch?v=lyZQPjUT5B4
Sorting Preview: Bubble Sort

- CardDeck used BubbleSort to sort the deck
- 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?
  - $O(n^2)$
- Space complexity?
  - $O(n)$ total (no additional space is required)
## Sorting Preview: Insertion Sort

<table>
<thead>
<tr>
<th>5</th>
<th>7</th>
<th>0</th>
<th>3</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Sorting Preview: 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 substantially sorted
- Time complexity
  - $O(n^2)$
- Space complexity
  - $O(n)$
## Sorting Preview: Selection Sort

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>3</th>
<th>27</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>16</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>5</td>
<td>16</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>11</td>
<td>16</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

- **Time Complexity:**
  - $O(n^2)$

- **Space Complexity:**
  - $O(n)$
Sorting Preview: Selection Sort

- Similar to insertion sort
- Performs worse than insertion sort in general
- 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

```java
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

```java
// 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

```java
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)

```java
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
- So, A[0]…A[last] are sorted.
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

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);
}
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;
Merge Sort

• A *divide and conquer* algorithm

• Merge sort works as follows:
  • If the list is of length 0 or 1, then it is already sorted.
  • Divide the unsorted list into two sublists of about half the size of original list.
  • Sort each sublist recursively by re-applying merge sort.
  • Merge the two sublists back into one sorted list.

• 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]` \(\text{split}\)
- `[8 14] [29 1] [17 39] [16 9]` \(\text{split}\)
- `[8] [14] [29] [1] [17] [39] [16] [9]` \(\text{split}\)
- `[8 14] [1 29] [17 39] [9 16]` \(\text{merge}\)
- `[1 8 14 29] [9 16 17 39]` \(\text{merge}\)
- `[1 8 9 14 16 17 29 39]` \(\text{merge}\)
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

• 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

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 case: \( k \leq 1 \): 0 comparisons: \( 0 < 1 \cdot 2^1 \checkmark \)

• Induction Step: Suppose true for all integers smaller than \( k \). Let \( T(k) \) be \# of comparisons for \( 2^k \) elements. Then

  \[ T(k) \leq 2^k + 2 \cdot T(k-1) \leq 2^k + 2(k-1)2^{k-1} \leq k \cdot 2^k \checkmark \]
Merge Sort

- Unlike Bubble, Insertion, and Selection sort, Merge sort is a divide and conquer algorithm
  - Bubble, Insertion, Selection sort complexity: $O(n^2)$
  - Merge sort complexity: $O(n \log n)$
- 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.

<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>
Recall Merge Sort

```java
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);
}
```
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”
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 $\leq 7$
  - 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><strong>Bubble</strong></td>
<td>Worst: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Best: $O(n)$ - if “optimiazed”</td>
<td></td>
</tr>
<tr>
<td><strong>Insertion</strong></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><strong>Selection</strong></td>
<td>Worst = Best: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td><strong>Merge</strong></td>
<td>Worst = Best: $O(n \log n)$</td>
<td>$O(n) : 2n + c$</td>
</tr>
<tr>
<td><strong>Quick</strong></td>
<td>Average = Best: $O(n \log n)$</td>
<td>$O(n) : n + c$</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.