CSE 2123
Sorting

Jeremy Morris
Problem Specification: Sorting

- Given a *list* of values, put them in some kind of sorted order
  - Need to know:
    - Which order?
      - Increasing?
      - Decreasing?
    - What does sorted order mean?
      - integers, doubles – numerical order
      - Strings, characters – alphabetical order
      - Other objects? Depends on the object
        - StudentGrade – By name? By grade?
Problem Specification: Sorting

- Note that values are stored in a *list*
  - Need to use a data structure where *order* is important
    - Sets, Maps – no real concept of “ordering”
  - Additionally, these algorithms use a data structure where *random access* is important
    - We will use arrays and ArrayLists for our examples
      - Random access
      - Order is obvious
Problem Specification: Sorting

- For our lecture examples, we will make a few assumptions:
  - Underlying collection type: simple array
  - Items to be sorted: integers
  - Order to be sorted: increasing numerical order

- Note that all of the sorting algorithms are extensible to other data types, collection types, and decreasing order
  - Need only minor modifications
Basic Sorting Tools

- All sorting algorithms use these two basic ideas:
  - **Swap**
    - Switch two values in the array
      ```
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      ```
  - **Compare**
    - Determine if two values are out of order
      ```
      (a[i] < a[j])
      (a[i] > a[j])
      ```
Swap – Java Code

```java
public static void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```
Comparing Sorting Algorithms

- How can we tell if one algorithm (A) is better than another one (B)?
  - Count how many swaps and comparisons it takes A to sort a list
  - Count how many swaps and comparisons it takes B to sort a list
  - Which one took fewer steps?
  - To be accurate, we need to consider worst case performance for both algorithms
Bubble Sort

Algorithm:

1. Start from the left side of the list
2. Visit each adjacent pair of list items in left-to-right order and swap if their values are out of order
3. Go back to step 1

- A single execution of steps 1 and 2 is called a pass
- When do we stop?
- Example: 6 8 7 5 4 3
At pass $i$ the following property holds:

- All entries from $a[(n - i +1)] \ldots a[n]$ are sorted, where $n$ is the index of the last element in the list.

- $i = 2$
- $n = 5$ (index starts at zero!)
- Items between index $(n-i+1)=4$ and $n=5$ are sorted.

```
6  5  4  3  7  8
```

Unsorted | Sorted
public static void bubbleSort(int [] a) {

    for (int i=0; i<a.length-1; i++) {  // For each pass
        for (int j=0; j<a.length-1; j++) { // Traverse the list
            if (a[j]>a[j+1]) {  // if out of order
                swap(a,j,j+1);  // SWAP
                // - see “Tools” slide
            }
        }
    }
}
Can we stop this algorithm sooner?

- Think about our sort property
- The inner *for* loop only needs to traverse to the end of the currently unsorted portion of the list
- We can stop our sort sooner on each pass

What if we make no swaps on the portion that is currently supposed to be unsorted?

- Can we stop if we have a pass where we make no swaps?
Selection Sort

Algorithm:
1. Start from the first position in the list – call it $i$
2. Find the smallest value in the list starting from position $i$ – call it $j$
3. Swap the value at $i$ with the value at $j$
4. Make $i = i + 1$
5. Go back to step 2

- Here a pass is a single execution of steps 2-4
- When do we stop?
- Example: 6 8 7 5 4 3
Selection Sort – Sort Property

- At pass $i$ the following property holds:
  - All entries from $a[0] \ldots a[i-1]$ are sorted, where $n$ is the index of the last element in the list
  - $i = 2$
  - Items between index 0 and 1 are sorted

<table>
<thead>
<tr>
<th>Sorted</th>
<th>Unsorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4</td>
<td>7 5 8 6</td>
</tr>
</tbody>
</table>
Selection Sort – Java Code

```java
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) { // For each pass
        int minIndex = i;
        for (int j = i + 1; j < a.length; j++) { // Traverse the list
            if (a[j] < a[minIndex]) { // if list item smaller
                minIndex = j; // make it the minimum
            }
        }
        swap(a, i, minIndex); // SWAP current and min.
    }
}
```
Insertion Sort

Algorithm:
1. Start from the second position in the list – call it \( i \)
2. Find where in the list to the left of \( i \) to insert the value at \( i \)
3. Shift values to the right of the insertion point to insert this value
4. Make \( i = i + 1 \)
5. Go back to step 2

- Here a pass is a single execution of steps 2-4
- We will perform \( n-1 \) passes for an array of size \( n \)
- Example: 6 8 7 5 4 3
At pass $i$ the following property holds:

- All entries from $a[0] \ldots a[i]$ are sorted
- $i = 3$
- Items between index 0 and 3 are sorted
public static void insertionSort(int[] a) {

    for (int i=1; i<a.length; i++) { // For each pass
        int insert = a[i];
        int scan = i;
        while (scan>0 && a[scan-1]>insert) {
            // Traverse the sorted side of the list to find the
            // insertion point
            a[scan]=a[scan-1];
            scan=scan-1;
        }
        a[scan]=insert; // insert the unsorted value
    }
}
Quicksort

Algorithm:
1. Select a pivot element
2. Partition the list around the pivot element
   1. Move elements less than the pivot to the left of the pivot (unsorted)
   2. Move element greater than the pivot to the right of the pivot (unsorted)
3. Recursively call quicksort on the left and right lists

Example: 6 8 7 5 4 3
public static void quickSort(int [] a, int start, int end) {

    if (start<end) { // general case
        int pivot = partition(a, start, end);

        // sort left sublist
        quicksort(a,start,pivot-1);

        // sort the right sublist
        quicksort(a,pivot+1,end);
    }
}
public static int partition(int[] a, int start, int end) {

    int pivot;
    int endOfLeft;
    int midIndex = (start+end)/2;

    swap(a,start,midIndex);
    pivot=a[start];
    endOfLeft=start;
    for (int i=start+1; i<=end; i++) {
        if (a[i]<pivot) {
            endOfLeft=endOfLeft+1;
            swap(a,endOfLeft,i);
        }
    }
    swap(a,start,endOfLeft);
    return endOfLeft;
}
Mergesort

Algorithm:
1. Partition the list at the middle to define two sublists
2. Recursively sort each sublist
3. Merge the two sorted sublists into one list

- What is our base case? General case?

- Example: 6 8 7 5 4 3
public static int[] mergeSort(int[] a, int start, int end) {

    if (start<end) { // general case
        int mid=(start+end)/2;
        int[] left = mergeSort(a,start,mid);
        int[] right = mergeSort(a,mid+1,end);
        return merge(left, right);
    }
    else {
        int[] arr = new int[1];
        arr[0] = a[end];
        return arr;
    }
}

Mergesort – Java Code
Merge Step

- Simple case – compare both values

```
mergeSort {6, 5}
```

```
mergeSort {6}
```

```
mergeSort {5}
```

```
merge {6} {5}
```

```
{5, 6}
```
Merge Step

- Large lists – traverse sublists and compare values

```
mergeSort
{6, 2, 5, 1}

mergeSort
{6, 2}
mergeSort
{5, 1}

merge
{2,6} {1,5}

1 < 2 : {1}
2 < 5 : {1,2}
5 < 6 : {1,2, 5}
6 : {1,2, 5, 6}
```
Merge Step

- Think of it as combining two stacks of cards
  - Remove lowest card shown on top and place it on new stack

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Merge Step

- Think of it as combining two stacks of cards
  - Remove lowest card shown on top and place it on new stack

\[
\begin{array}{ccc}
\text{A} & \text{B} & \text{A} & \text{B} \\
2 & 1 & 2 & 3 \\
6 & 3 & 6 & 4 \\
7 & 4 & 7 & 5 \\
8 & 5 & & 8 \\
\end{array}
\]
Merge Step

- Think of it as combining two stacks of cards
  - Remove lowest card shown on top and place it on new stack

```
A  B      A  B   A  B   A  B
2  1      2  3   6  3
6  3 → 6  4 → 7  4 →
7  4      7  5   8  5
8  5      8
```
Merge Step

- Think of it as combining two stacks of cards
  - Remove lowest card shown on top and place it on a new stack

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Merge Step

- Think of it as combining two stacks of cards
  - Remove lowest card shown on top and place it on new stack

```
A  B      A  B   A  B   A  B   A  B
2  1      2  3   6  3   6  4   6  5
6  3      6  4   7  4   7  5   7
7  4      7  5   8  5   8
8  5      8
```
**Merge Step**

- Think of it as combining two stacks of cards
  - Remove lowest card shown on top and place it on a new stack

```
A  B  A  B  A  B  A  B  A
2  1  2  3  6  3  6  4  6
6  3  6  4  7  4  7  5  7
6  4  7  5  7  8  8  8  8
7  5  8  8
```

public static int[] merge(int[] left, int[] right) {
    int lIndex=0;
    int rIndex=0;
    int newIndex=0;
    int[] list = new int[left.length+right.length];
    while (lIndex<left.length && rIndex<right.length) {
        if (left[lIndex]<=right[rIndex]) {
            list[newIndex]=left[lIndex];
            lIndex=lIndex+1;
        } else {
            list[newIndex]=right[rIndex];
            rIndex=rIndex+1;
        }
        newIndex=newIndex+1;
    }
    while (lIndex<left.length) {
        list[newIndex]=left[lIndex]; lIndex=lIndex+1; newIndex=newIndex+1;
    }
    while (rIndex<right.length) {
        list[newIndex]=right[rIndex]; rIndex=rIndex+1; newIndex=newIndex+1;
    }
    return list;
}