Sorting

- Sorting is common to many applications
  - Part of formatting, organizing
  - May be required for efficient searching
- Many different algorithms to perform sorting
  - Some more efficient with space (if data is extremely large)
  - Some more efficient with time (if speed is critical)
  - Some more efficient with certain patterns of data (e.g., slightly sorted)

Array Sorting

- We'll start with looking at some ways of sorting an array
  - Problem statement: given an array, re-order the elements of the array in increasing order
- Functionality required is
  - Ability to compare two elements in array, both for equality and order
  - Ability to exchange two elements in array (swap)
- Encapsulate in a class with sorting methods
Array Sorting

- Consider the easy cases:
  - If the array is empty or has just one element, then it can be considered to be sorted
  - If the array has two elements, then compare the elements
    - If they are in increasing order, then the array is sorted
    - If not, then swap the two elements
- For more than two elements
  - Decompose into smaller problems

Decomposition of Sorting Problem

- Divide array into smaller pieces
  - Suppose n is length of array, and 0 ≤ i < n
  - We have two smaller arrays: a[0]...a[i-1] and a[i]...a[n-1]
  - Note that either of these pieces may be empty
- If we try all values of i from 0 to n-1, that would be a straightforward iterative approach
  - Implement with a loop on i
  - Selection Sort and Insertion Sort are both iterative approaches that work through the array this way
  - Both maintain a[0]...a[i-1] is sorted as an invariant
- Other divisions (e.g., divide in half) would likely be a recursive approach
  - Implement by recursive calls on partition, re-computing next partition each time
Selection Sort

- Finds the minimum in the unsorted part $a[i]...a[n-1]$ and swaps it with $a[i]$
  - Additional invariant: all of the values in $a[0]...a[i-1]$ are less than all the values in $a[i]...a[n-1]$

Note at $i^{th}$ step, there are $n-i-1$ comparisons to find the minimum, for a total of $n-1 + n-2 + ... + 1 + 0$, which is $O(n^2)$

One swap for each step, which is $O(n)$

---

Insertion Sort

- Takes first element in unsorted part $a[i]...a[n-1]$ and inserts it in proper position of $a[0]...a[i-1]$
  - Does the insertion by comparing and swapping with $a[i-1]$, then $a[i-2]$ until in correct position

Note at $j^{th}$ step, there are possibly $n-i-1$ comparisons and swaps, which is $O(n^2)$
Iterative Sorting Complexity

<table>
<thead>
<tr>
<th>Sort Type</th>
<th>worst</th>
<th>average</th>
<th>best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>comparisons</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

- Selection sort has same behavior for all cases

<table>
<thead>
<tr>
<th>Sort Type</th>
<th>worst</th>
<th>average</th>
<th>best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>comparisons</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

- Insertion sort is faster for "almost sorted" data

Recursive Sorting Algorithms

- Instead of decomposing the sorting problem by looping, use a divide and conquer approach with recursion
- Classic algorithms: Merge Sort and Quick Sort
  - Base case is segment of size 0 or 1
  - Partition the array into segments of size at least 1
  - Call the sort function recursively on partitions
  - Combine the results to preserve invariant
Merge Sort

- Idea is to divide (in half), recursively get halves sorted, then merge into single sorted list
- To merge two sorted halves of a segment
  - Create a temporary copy of the segment
  - Move through the two halves, taking the smaller of the first element of each half and copying it over the original segment

```
3   4   27  48  23  34  41  45  89
4   8   23  41  45   3   7  34  89
3  8  23  41  45   3   7  34  89
4   8  23  41  45  7  34  89
3  4  23  41  45   3   7  34  89
8  23  41  45  7  34  89
3  4  7  41  45   3   7  34  89
8  23  41  45  34  89
3  4  7  8  45   3   7  34  89
23  41  45  34  89
3  4  7  8  23  34  7  34  89
41  45  34  89
3  4  7  8  23  34  41  34  89
45  89
3  4  7  8  23  34  41  45  89
```

Implementation of MergeSort

```java
public void mergeSort() {
    recMergeSort(0, a.length-1);
}

private void recMergeSort(int low, int high) {
    if (low == high) return;  // Base case, it's sorted
    else {
        recMergeSort(low, (low+high)/2);  // Sort first half
        recMergeSort(1+(low+high)/2, high); // and second
        merge(low, high);  // Combine
    }
}
```
Implementation of merge

```java
private void merge(int low, int high) {
    int[] temp = new int[high-low+1];
    for (int k = low; k <= high; ++k)
        temp[k-low] = a[k];
    int i = 0, j = 1+(temp.length-1)/2;
    for (int k = low; k <= high; ++k) {
        if (j >= temp.length)            a[k] = temp[i++];
        else if (i > (temp.length-1)/2)  a[k] = temp[j++];
        else if (temp[i] < temp[j])      a[k] = temp[i++];
        else                             a[k] = temp[j++];
    }
}
```

MergeSort Complexity

- Complexity of merge in terms of size of interval (let k = high – low + 1)
  - Create new array (of size k) and copy elements over: c·k
  - For each element, one comparison and copy: c·k
  - Cost of merge is 2·c·k
- recMergeSort recurrence relation
  - T(0) = T(1) = c
  - T(n) = 2T(n/2) + 2cn = 2(2T(n/4) + 2cn/2)+2cn = 4T(n/4) + 2cn +2cn = 8T(n/8) + 2cn + 2cn +2cn = ... = 2cn(1+1+...+1)  [log₂n times]
- So MergeSort is O(n·log n) time
  - Should also consider space requirements – O(n)
Quick Sort

- Idea is to partition around a "pivot" element, rearranging so that all values less than pivot are to left, all values greater to right (i.e., pivot is in right place)
  - Recursively sort segments to left and right (excluding pivot)
- Partitioning a segment
  - Choose first value as pivot value
  - Move pivot to right if it is greater
  - Move value to end if pivot is smaller
  - Return index of pivot

```
4 8 41 45 23 7 34 89 34 8 41 45 23 7 34 89 3
pivot value: 4

4 3 41 45 23 7 34 89 8
pivot value: 4

3 4 41 45 23 7 34 89 8
pivot value: 4

3 4 89 45 23 7 34 41 8
pivot value: 4

3 4 34 45 23 7 89 41 8
pivot value: 4

3 4 7 45 23 34 89 41 8
pivot value: 4 pivot index: 1

3 4 45 23 7 34 89 41 8
pivot value: 45

3 4 23 45 7 34 89 41 8
pivot value: 45

3 4 23 7 34 45 89 41 8
pivot value: 45

3 4 23 7 34 45 8 41 89
pivot value: 45
pivot index: 7

3 4 23 7 34 8 45 41 89
pivot value: 23

3 4 7 23 34 8 41 45 89
pivot value: 23

3 4 7 23 41 8 34 45 89
pivot value: 23

3 4 7 23 8 41 34 45 89
pivot value: 23
pivot index: 4

3 4 7 8 23 41 34 45 89
pivot value: 7

3 4 7 8 23 41 34 45 89
pivot value: 7

3 4 7 8 23 34 41 45 89
pivot value: 41

3 4 7 8 23 34 41 45 89
pivot value: 41

3 4 7 8 23 34 41 45 89
pivot value: 41
pivot index: 7
```

Implementation of QuickSort

```java
public void quickSort() {
    recQuickSort(0, a.length-1);
}

private void recQuickSort(int low, int high) {
    if (low >= high)  // empty segment or single item
        return;
    else {
        // Partition by pivot
        int pivot = partition(low, high);
        // Sort the parts
        recQuickSort(low, pivot-1);
        recQuickSort(pivot+1, high);
    }
}
```
Implementation of partition

```java
private int partition(int low, int high) {
    int pivotVal = a[low];
    while (low < high) {
        if (a[low+1] <= pivotVal) {
            // Move the pivot beyond smaller value
            a[low] = a[low+1];  ++low;  a[low] = pivotVal;
        } else {
            // Move the larger value to end
            int highVal = a[high];
            a[high] = a[low+1]; a[low+1] = highVal; --high;
        }
    }
    return low;
}
```

QuickSort Complexity

- Complexity of partition in terms of size of interval (let $k = high - low + 1$)
  - For each element, one comparison and swap: $c \cdot k$
  - Cost of partition is $c \cdot k$, i.e., $O(k)$ time
- recQuickSort recurrence relation
  - $T(0) = T(1) = c$
  - $T(n) = T(n-i) + T(i) + cn$
  - QuickSort average case is $O(n \cdot \log n)$ time
  - Worst case is $O(n^2)$
  - In place partitioning means no extra space