Sorting, Part 1

1. Last time:
    a. How to compute asymptotic bounds for some common cases
    b. Taste of proving general bounds (*take Math 251 and CS 256 for more of this!*)
2. Course schedule details:
    a. New homework 2 due Monday
    b. Exam next week in lab (there will be a new lab assignment, but it is easy)
    c. Two review sessions scheduled for next week, plus regular office hours

3. "Sorting" means putting values in order
    a. Examples. Sort these values by hand:
        i. {5, 1, 6, 2, 3, 4}

        ii. {1, 2, 3, 4, 5, 6}

        iii. {"brown", "a", "cat"}

    b. Motivating applications
    c. What is "in order"?
4. **java.util.Comparator**

```
public class IntegerComparator implements java.util.Comparator<Integer> {
    public int compare(Integer a, Integer b) {
        if      (a.intValue() > b.intValue())    return 1;
        else if (a.intValue() == b.intValue())   return 0;
        else                                     return -1;
    }
}
```

5. Merge Sort
    a. Merge two already-sorted lists
    b. Building a sort out of the merge operation

Next time: we'll analyze mergeSort…and see if we can sort even faster.

Solutions to Homework 1, Questions A and B

A. Bound the space required by the following method:

```
public static String str(String x) {
      if (x.length() <= 1)          return x;
      else                           return x + str( x.substring(1) ) + str(x.substring(2));
}
```

First, consider the space required for the final result string. Let n be the length of the input string. Let S_n be the length of the output string for that input. We can see immediately from the recursive case above that:

$$S_n = n + S_{n-1} + S_{n-2}. \qquad (1)$$

Because the recursion strictly makes the argument passed along shorter using substring, we know that

$$S_{n-1} \geq S_{n-2}. \qquad (2)$$

Furthermore, because each recursive call (except the last) concatenates at least one character onto its argument, we know that

$$S_{n-1} \geq n \quad \text{for} \quad n > 0. \qquad (3)$$

Subsituting (2) and (3) into (1) allows us to make the conservative upper bound:

$$S_n \leq S_{n-1} + S_{n-1} + S_{n-1} \qquad (4)$$
$$S_n \leq 3S_{n-1} \qquad (5)$$
$$S_n = O(3^n) \qquad (6)$$

This is conservative. It is possible, but much more difficult to solve for a tight bound (I belive that the tightest bound is somewhere around 1.3^n). The bound proved here is useful in that it identifies that the space required is exponential in the input. Now, that's only the final string computed. What about all of the others that were produced along the way? I'm leaving that for you to think about some more because it appears on this week's homework in another form…

B. Bound the execution **time** of the following Java method:

```
public static float sinc(float n) {  return Math.sin(n) / n; }
```

All operations except for sin(n) are clearly constant time (for 32-bit floats). So the problem is equivalent to finding the time bound on sin(n). There are many reasonable theories you could put forth for how that is computed. One is to approximate sin using the Taylor series

$$\sin n = n - \frac{n^3}{3!} + \frac{n^5}{5!} - \frac{n^7}{7!} + \cdots,$$

including only as many terms as required to make the series converge to the smallest representable floating point number. Regardless of how many terms that is, it is constant independent of n (again, assuming that a float has a fixed number of bits), so sin can be computed in O(1).

Another theory is that the computer stores a huge table of precomputed sines. Since sine is cyclic, a finite length table can store the entire function to a fixed precision. That requires a huge amount of space but only O(1) time. So, how does Java actually compute sin? It depends on the underlying processor architecture. Some processors use the Taylor series and some use small tables extended by trigonometric identities.