```
                              CS 4: Lecture 28
                           Wednesday, May 3, 2006
```

PERFORMANCE
===========
As you've probably seen in the optimizer in your Lunar Lander code, computers can sometimes be slow. Working programmers spend a lot of time designing their programs to run fast. Sometimes you can speed up a program by using a more sophisticated algorithm. For example, in Lecture 7 we saw that you can often perform a simulaton in many fewer timesteps if you use the trapezoid rule instead of Euler's method, with the same accuracy.

Java provides a method called System.currentTimeMillis() that you can use to measure how long a portion of your code takes.

```
  long startTime = System.currentTimeMillis();
  ... [do some work] ...
  long workTime = System.currentTimeMillis() - startTime;
```

Now workTime is the amount of time spent doing the work, measured in milliseconds.

Let's compare the time required to perform a matrix multiplication operation, in Java and Matlab. Suppose you have two nxn matrices A and B, and you want to compute C = AB. You need to compute

```
         n
  C   = sum A   B
   ij   k=1  ik  kj
```

for every i, j in the range 1...n.

In Java, you could use the following loop.

```
  public static double[][] matmul (double[][] A, double[][] B) {
    int n = a.length;
    double[][] C = new double[n][n];
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
          C[i][j] += A[i][k] * B[k][j];
        }
      }
    }
    return C;
  }
```

In Matlab, the same loop would look like this.

```
  for i = 1 : n
    for j = 1 : n
      for k = 1 : n
        C(i, j) = C(i, j) + A(i, k) * B(k, j);
      end
    end
  end
```

But, remember that Matlab has matrix operations built in. So it's much simpler to just write

```
  C = A * B;
```

How long do these matrix mulitplications take? If n = 360,
- the triply-nested Java loop takes about 1.7 seconds.
- The triply-nested Matlab loop takes about 400 seconds.
- The built-in Matlab matrix multiply takes less than a tenth of a second.

Why? Matlab is an interpreted language, and the interpreter runs quite slowly compared to the partly-compiled Java code. But Matlab's built-in matrix multiply runs highly optimized code, which the numerical experts at Matlab have slaved over to make it as fast as possible. It's probably written directly in machine language, and optimized for the specific microprocessor you're using.

Big-Oh Notation (upper bounds on running time)

All three matrix multiplications have one thing in common, though. They execute n^3 iterations of the inner loop (which performs one multiplication and one addition). For really large values of n, the running time is approximately c n^3, where c is a constant...but the constant c is different for each of the three implementations.

We say they all run in O(n^3) time. This is called _big-oh_notation_.

We know the constant c will change if we buy a faster computer, or use a different language or compiler. Big-oh notation expresses the speed of an algorithm independently of a specific implementation on a specific machine, ignoring constant factors (which get smaller and smaller as technology improves). One of the nice things about an algorithm's big-oh running time is that you can figure it out by counting how many operations the algorithm performs. It doesn't matter what computer you use.

Suppose an algorithm for processing a retail store's inventory takes (10,000 + 10 n) ms to process n transactions. That's
  - 10,000 milliseconds to read the initial inventory from disk, and then
  - 10 milliseconds to process each transaction (items acquired or sold).

We say this program runs in O(n) time. Why? Even though 10,000 >> 10, the "10 n" term will be more important if the number of transactions is very large. Big-oh notation ignores all the terms except the term that grows fastest as n increases. Big-oh is an example of "asymptotic notation", so called because it's concerned with how the running time behaves as n -> infinity.

Let n be the size of a program's _input_ (in bits or data words or whatever).
Let T(n) be a function that specifies the algorithm's running time, given an
  input of size n.
Let f(n) be another function--preferably a simple function like f(n) = n.

Formally, T(n) is in O(f(n)) if

```
               T(n)
      lim      ---- < infinity.
  n->infinity  f(n)
```

For example,

```
              10,000 + 10n
      lim     ------------ = 10,
  n->infinity       n
```

so 10,000 + 10n is in O(n). But

```
              10,000 + 10n
      lim     ------------ = infinity,
  n->infinity       1
```

so 10,000 + 10n is not in O(1).

Efficiency

Often, the best way to make a program faster is to use an algorithm with a
better asymptotic running time.

(1) Searching for a number in a sorted array of ints takes O(n) time if you
search the whole array from left to right, but only O(log n) time if you
use binary search.

(2) Determining all the prime numbers in 2...n takes O(n^2) time if you do it
the easy way, trying every number against every possible divisor. The
Sieve of Eratosthenes, which you implemented for the midterm, takes
O(n log n) time.

(3) Computing a Fibonacci number using the recursive algorithm you used in lab
takes

```
       1 + sqrt(5)  n    ~         n
    O( ( ----------- )  )  ~  O(1.618 )
             2
```

time, which grows exponentially--that means _incredibly_ quickly. But the
iterative algorithm takes only O(n) time.

ALGORITHM ANALYSIS
==================
Problem #1: Given an array of n integers, find the smallest.
Algorithm #1: Maintain a variable "min" that stores the smallest integer
scanned so far.
Set "min" to the first element of the array.
Scan through the rest of the array, updating "min" as necessary.

The running time is in O(n), because all we do is scan the whole array once.

Problem #2: Given a set of p points, find the pair closest to each other.
Algorithm #2: Calculate the distance between each pair; return the minimum.

There are p (p - 1) / 2 pairs, and each pair takes constant time to examine.
Therefore, the running time is in O(p^2).

Hint: The code for this algorithm gives a strong hint, because it contains
a doubly-nested loop, with both loops iterating through input points.

```
double minDistance = point[0].distance(point[1]);

/* Visit a pair (i, j) of points. */
for (int i = 0; i < numPoints; i++) {
  /* We require that j > i so that each pair is visited only once. */
  for (int j = i + 1; j < numPoints; j++) {
    double thisDistance = point[i].distance(point[j]);
    if (thisDistance < minDistance) {
      minDistance = thisDistance;
    }
  }
}
```