<u>Introduction and Complexity</u>

data structure

- a way of storing data in a computer so that it can be "used" efficiently
- simple examples of data storage (not actual structures) you've seen
  - short, int, long, float, double, char, boolean (bool in C++)
- the bottom line is that we have a feedback loop (it's like this in life too)
  - input, process, output
  - how do we process the input efficiently to generate the "right" output?
  - we usually need to manipulate (search, sort, change, etc) data
  - how can we store this data in the "computer" to make processing better?
  - this is, essentially, what this class is about

first, we need to discuss the issue of complexity

- how do we measure the performance/efficiency of an algorithm?
  - how long does it take to run?
  - what's its the space/size/memory usage?
  - we use the "big-o" notation which describes an upper bound of performance
- the execution time of an algorithm goes up as the amount of data goes up
  - but we throw out how much data we have
  - and we'll call algorithms good for small amounts of data no matter the algorithm
- different processors may execute an algorithm in different times
  - but we'll throw out how fast a CPU is
- different languages may affect algorithm speed
  - but we'll throw out the language and call them all "the same"
- so we only want to look at the algorithm itself
  - particularly at loops since most time is spent there

suppose the time (or number of steps) it takes to complete a problem of size *n* were:

$$T(n) = 4n^2 - 2n + 2$$

as *n* grows large (this is what we care about), the n^2 term will dominate

so all other terms can be ignored

e.g. when $n = 500$, $4n^2$ is 1000 times as large as $2n$

so it's safe to ignore it (and the 2 doesn't matter either)

e.g. (the complexity for this one is n^3)

```
for (k=0; k<n/2; k++)
{
        ...    // this statement occurs n/2 times
        for (j=0; j<n*n; j++)
        {
                ...    // this statement occurs n*n*n/2 = n³/2 times
        }
}
```

another e.g. (the complexity for this one is n^2)

```
for (k=0; k<n/2; k++)
{
        ...    // n/2
}
for (j=0; j<n*n; j++)
{
        ...    // n*n
}
```

yet another e.g. (the complexity for this one is $\log_2 n$)

```
k = n;
while (k > 1)
{
       ...
       k /= 2;       // integer division - log2n
}
```

note that $\log_2 n$ implies division by two
and that $\log_3 n$ implies division by three (and so on)

polynomial algorithms

$$O\left(a_m n^m + a_{m-1} n^{m-1} + \ldots + a_2 n^2 + a_1 n + a_0\right)$$

exponential algorithms

$$O\left(a^n\right)$$

logarithmic algorithms

$$O\left(\log_b n\right)$$

remember the log rules?

$$\log_2 4 = 2 \;\equiv\; 2^2 = 4$$
$$\log_e 2.718 = x$$
$$\log_2 1000 \approx x$$
$$\log_b(mn) = \log_b(m) + \log_b(n)$$
$$\log_b\left(\frac{m}{n}\right) = \log_b(m) - \log_b(n)$$
$$\log_b(m^n) = n \log_b(m)$$
$$\log \rightarrow \log_{10}$$
$$\ln \rightarrow \log_e$$
$$\lg \rightarrow \log_2$$
$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)}$$

common dominant terms

n dominates $\log_b n$; b is often 2
$n \log_b n$ dominates n ; b is often 2
n^m dominates n^k when $m > k$
a^n dominates n^m for any values of a and m greater than 1

the more influential term dominates

we drop everything else (constants, etc)

HOMEWORK complexity worksheet