1: Introduction to Data Structures & Complexity.

A **data structure** is a way of storing data in a computer so that it can be used efficiently. In your programming experience so far, you have used some data structures but may not have known that they were data structures. An array is an example of a data structure. The elements are arranged in contiguous memory locations so that they can be accessed, stored, and manipulated quickly. Other types of data structures include *sets*, *unions*, *records*, *graphs*, *trees*, etc. Do not confuse data types and data structures. Think of data types as atoms and data structures as molecules. Examples of data types are `int`, `short`, `long`, `float`, `double`, `char`, `boolean`, etc.

So we've said that we store data in data structures so that it can be *manipulated efficiently*, but what exactly does that mean? Manipulation usually means searching, sorting, changing, etc. We normally process some form of input efficiently to generate the "right" output. Data structures allow us to store this data in a way that makes the processing better.

Data structures is essentially what this class is about (as I have mentioned before on many occasions). We will spend the rest of this quarter looking at different commonly used data structures and how we can use them in our code.

Complexity

How do we measure the performance of an algorithm? Is it the time it takes to run? Or the space/size/memory that it requires? Well it turns out that it is both. Even so, we tend to describe the performance of an algorithm using speed of execution. We use what is referred to as the "big-O" notation to describe the upper bound of performance.

With most manipulations, the execution time will increase with the amount of data that we are manipulating. This means that a small amount of data will demand a shorter run time, while a larger amount of data will demand a longer run time.

Some of the difference between execution times can be attributed to device specific features such as processors and CPU speed, type of programming language, etc. But the Big-O notation seeks to describe algorithm performance independent of these specific features i.e. if you consider the algorithm and the algorithm only, how does it perform with this amount of data? Most of the time, algorithm statements are executed once and therefore would not take a considerable amount of time to be executed (especially if it is dealing with a large amount of data). However, there are statements that are executed multiple times and it is these parts of the algorithm that will dominate execution time. Statements that are executed multiple times are normally enclosed within a loop of some kind. The Big-O notation attempts to describe how many times those statements would be executed in terms of the size of the algorithm's input.

Many times this performance will be described in terms of a mathematical function whose argument is *n* and represents the size of the data. For example, suppose the time T (number of steps) it takes for a hypothetical algorithm to complete a problem of size n is given by the expression

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

Then we know that if our size is 1, it will take

$$T(1) = 4 - 2 + 2 = 4 \text{ steps}$$

If our size is 10, it will take

$$T(10) = (4 * 10^2) - (2 * 10) + 2 = 382 \text{ steps}$$

If our size is 100, it will take

$$T(100) = (4 * 100^2) - (2 * 100) + 2 = 39802 \text{ steps}$$

and if our size is 1000, it will take

$$T(1000) = (4 * 1000^2) - (2 * 1000) + 2 = 3999802 \text{ steps}$$

You will have noticed that while our input was only increasing by a factor of *10*, the number of steps was increasing by a factor of *100*...ish. This is because the dominant term of $T(n)$ is n^2 and so we can describe the complexity of this hypothetical algorithm as being $O(n^2)$.

The big-O notation can then be used to provide an estimate on the amount of time (or number of steps) that an algorithm would require for a given data size. For example given $O(n^2)$ where n is *1000*, the algorithm would take approximately:

$$O(1000) \approx 1000^2 = 1000000 \text{ steps}$$

The number of steps can then be translated to time if you know how long the average step would take on a specific system.

The trick for the big-O notation is to figure out an expression for the number of times statements will be repeated in an algorithm (T(n)) paying special attention to the statements that occur in loops that are dependent on the data size. Once you have that expression, you reduce the entire expression to a simplified expression made up of the dominant term of the original expression.

Let's look at a few examples of code and see if we can deduce their complexities.

```
for (k=0; k<n; k++)
{
      …           // How many times will this statement be executed?
}
```

Any line in the `for` loop above will be executed n times. Therefore $T(n) = n$ and its complexity is $O(n)$. What about the one below.

```
for (k=0; k<n; k++)
{
      …           // How many times will this statement be executed?
      for (j=0; j<n; j++)
      {
            …     //What about this statement?
      }
}
```

$T(n) = n^2 + n$ which means it is $O(n^2)$.

Those were fairly straight forward. What about these ones?

```
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
      }
}
```

$T(n) = n^3/2 + n$ which means it is *$O(n^3)$*.

```
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 = n² times
}
```

$T(n) = n^2 + n/2$ which means it is *$O(n^2)$*.

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

$T(n) = log_2 n$ which means it is *$O(log_2 n)$*.

$log_2 n$ implies division by 2.
$log_3 n$ implies division by 3.
$log_4 n$ implies division by 4...and so on.

The way we moved from T(n) to Big-O might have seemed confusing so here are some things to remember. We ALWAYS identify the dominant (or more influential) term, and drop everything else (constants too).

But how do we find out the dominant term? Well it changes from case to case but here are a few rules of thumb.

- *n* dominates *$log_b n$* (*b* is often *2*)
- *$nlog_b n$* dominates *n*
- *n^m* dominates *n^k* when *$m>k$*
- *a^n* dominates *n^m* for any values of *a* and *m* greater than *1*.