**Chapter 2. Understanding Time/Space Complexity**

**Complexity Analysis:**

How much time does this algorithm need to finish?

How much space does this algorithm need for its computation? (RAM or secondary storage).

**Big-O Notation:**

Or “algorithmic efficiency”. Gives a measurement of the complexity of the algorism’s worst case scenario. EG trying to find a number by checking each number in a list, and target number is at the very end of the list.

O = represents the upper bound of asymptotic complexity.

N = The size of the input.

Complexities ordered from smallest to largest:

* Constant Time: 0(1)
* Logarithmic Time: 0(log(n))
* Linear Time: 0(n)
* Linearithmic Time: 0(nlog(n))
* Quadratic Time: 0(n2)
* Cubic Time: 0(n3)
* Exponential Time: 0(bn), b > 1
* Factorial Time: 0(n!)

Another explanation here:  
  
Big O notation seeks to describe the *relative complexity* of an algorithm by reducing the growth rate to the key factors when the key factor tends towards infinity. For this reason, you will often hear the phrase *asymptotic complexity*. In doing so, all other factors are ignored. It is a **relative representation of complexity**.

<http://www.cforcoding.com/2009/07/plain-english-explanation-of-big-o.html>

**Big-O Examples:**

Constant Time

The following run in constant time 0(1). Because they don’t depend on n (size of the input) at all.

Example1:

a = 1, b =2;

c = a+ b\*5;

Doing a mathematical formula is run in a constant amount of time.

--------------------------------

Example2:

i = 0;

while (i < 11){

i++;

}

This loop doesn’t depend on n, so is run in a constant amount of time.

IE as the input size gets larger, this loop will still run in the same amount of time.

Linear Time

The following run in Linear Time O(n). With respect to the input size n.

Because we do a **constant** amount of work, **n times**.

Example1:

i= 0;

While(i< **n**){

i++;

}

*F(n) = n*

*O(f(n)) = O(n)*

--------------------------------

Example2:

i = 0;

while (i < n){

i = i + 3;

}

Incrementing by 3 means we finish the loop 3 times faster than just incrementing by 1.

So *f(n) = n/3* instead.

*F(n) = n/3*

*O(f(n)) = O(n)*

Quadratic Time

n work, done n times. n\*n = O(n2)

Example1:

For (int i=0; i<n; i=i+1)

For (int j=0; j<n; j=j+1)

*F(n) = n\*n = n2*

*O(f(n)) = O(n2)*

--------------------------------

Example2:

For (int i=0; i<n; i=i+1)

For (int j=i; j<n; j=j+1) *- j initialised by i, not 0*.

Since I goes from [0,n) the amount of looping done is directly determined by what i is.

If i=0, we do n work. If i=1, we do n-1 work. If i=2, we do n-2 work etc.