Asymptotic Complexity and Big-O

Definition 1.0. If for two function $f(n)$ and $g(n)$, $f(n) \in O(g(n))$ if and only if there exists some $n_0 \in \mathbb{N}$ and $c \in \mathbb{R}^+$, that for all $n \in \mathbb{N}$ such that $n > n_0$ $f(n) \leq c \cdot g(n)$.

Definition 1.1. If for two function $f(n)$ and $g(n)$, $f(n) \in \Omega(g(n))$ if and only if there exists some $n_0 \in \mathbb{N}$ and $c \in \mathbb{R}^+$, that for all $n \in \mathbb{N}$ such that $n > n_0$ $f(n) \geq c \cdot g(n)$.

Definition 1.2. Given a function $f(n)$ and $g(n)$, $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Definition 1.3. *Asymptotic Analysis* is when one determines the run time of an algorithm for a given set of parameters. Often times the analyses hold for when things approach infinity. A basic table in order of goodness is as follows

Function order	Description
$O(k)$	Constant (No Growth)
$O(\log n)$	Logarithmic (Low Rate of Increase with n)
$O(n)$	Linear (Goodish Stuff)
$O(n^2)$	Quadratic Time (This can be okay...)
$O(n^k)$	Polynomial Time (Bad stuff)
$O(k^n)$	Exponential (Please stop)
$O(n!)$	Factorial (You should just not run this)

There are generally two ways to approach finding the complexity of an algorithm. One can do it mathematically or through writing up a program and testing it. Often times it is better to do it mathematically because the performance may depend upon..

1. Variability - Different systems will run at different rates due to harware/software.
2. Portability - Changes based on implementations on machines.
3. Other programs running can change speed.
4. Choice of input - may miss worst case scenario or even best case.

hence the reason for an algorithm running slowly may get obfuscated.

To come up with the complexity of an algorithm mathematically, we must generate some system to relate runtime to the code created.

Some operations are considered constant, so they do not depend on the parameters given. Some examples and classes are as follows

1. Arithmetic (fixed with) operations. (Addition, subtraction, etc.)
2. Assignment. (x := 1)

3. Accessing an array element.

When analyzing algorithms, we often just lump all these together as a value of 1. If something has a remarkably larger time on it because lots of sub operations happen, then you may need to look into minimizing that operation.

To gague the performance of different controls in the program, look to this chart

Control Flow	Time
Consecutive Sequences	Total up all operations
Conditional Statemetns	Use the complexity of the larger branch
Loop	Number of iterations times complexity of the body of the block
Function Calls	Equivalent to the body of the function
Recursive Function Calls	Use a recurrence relation

For a recursive function, the recurrence relation is the function which represents the complexity of that function as a series of recursive calls. Consider the functions

```
int help(int* arr, int k, int start, int end)
{
    if (start == end) return 0;
    if (arr[(start + end) / 2] == k) return 1;
    if (arr[(start + end) / 2] ¡ k)
        return help(arr, k, start, (start + end)/2);
    return help(arr, k, (start + end)/2, end);
}

int binarySearch(int* arr, int len, int k)
{
    return help(arr, k, 0, len);
}
```

We define the recurrence relation in terms of the distance between start and end as

$$T(n) = \begin{cases} 1 & n == 1 \\ 10 + T(n/2) & else. \end{cases}$$

So $T(n) = 10 + T(n/2) = 20 + T(n/4) = 10k + T(n/2^k)$ for any $k \in \mathbb{N}$.
So for the base case of $n/2^k = 1$, $k = \log(n)$. Hence, $T(n) = 10\log(n)$.

Friday Jan 20 2017: Sorting Asymptotic Analysis

Exercise 1.4. Mergesort. Mergesort takes an array of length n, and splits it into subarrays of length n/2 and sorts those. An array of size 1 is sorted. Then you merge going back up. Example given below

Splitting Step

[8, 2, 9, 4, 5, 3, 1, 6]

[8, 2, 9, 4] [5, 3, 1, 6]

[8, 2] [9, 4] [5, 3] [1, 6]

[8] [2] [9] [4] [5] [3] [1] [6]

Merging Step

[2, 8] [4, 9] [5, 6] [1, 6]

[2, 4, 8, 9] [1, 3, 5, 6]

[1, 2, 3, 4, 5, 6, 7, 8]

And thus it is sorted

Proof. We define the recurrence function T as for all $n \in \mathbb{N}$,

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{2}) + n & n > 1. \end{cases}$$

For $n \in \mathbb{N}$ such that $n > 1$,

$T(n) = 2T(\frac{n}{2}) + n = 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + 2n = ...$

So for some $k \in \mathbb{N}$ for k is the number of recursion,

$T(n) = 2^k T(\frac{n}{2^k}) + kn.$

Consider $\frac{n}{2^k} = 1$, also known as the base case. This is due to this being where the recursion gets to the base case of $T(\frac{n}{2^k}) = T(1)$. This implies that $k = log(n)$.

Hence,

$$T(n) = 2^{log(n)} T(\frac{n}{2^{log(n)}}) + log(n) * n.$$

$$= nT(1) + nlog(n) = n + nlog(n).$$

Therefore, $T(n) \in O(nlog(n))$ and mergesort has complexity $O(nlog(n))$.

Consider the substitution method.

We assume that $T(n) \in O(nlog(n))$. Therefore, for some $c \in \mathbb{R}$,

$$c\ nlog(n) \geq 2T(\frac{n}{2}) + n$$

$$c\ nlog(n) \geq 2(\frac{n}{2})log(\frac{n}{2}) + n$$

$$c \geq \frac{n(log(n) - log(2)) + n}{nlog(n)}$$

$$c \geq \frac{log(n) - 1 + 1}{log(n)}$$

$$c \geq \frac{log(n)}{log(n)}$$

$$c \geq 1$$

So because there exists some $c \in \mathbb{N}$ such that $c \cdot nlog(n) \geq T(n)$, $T(n) \in O(nlog(n))$.

□