Orders of Growth and Intro to Asymptotics

1 An Introduction to Asymptotics

Now that we've discussed programming, we are going to focus on how to make our programs run well enough to work in the real world. In order to do this, we will discuss *asymptotics*, a way of measuring either the **Time Complexity** or **Space Complexity**. The formedr comes more into play in the next few chapters; however, we will discuss Space Complexity in Chapter 5 and 6 briefly.

2 Order of Growth

The speed at which a program runs can vary from computer to computer based off various factors such as processors. In order to measure speed at a uniform scale, we use asymptotics, which give tell us how a program runs as the input scales to an arbitrarily large number. It is extremely important to realize that no matter how powerful your computer is, your asymptotic runtime will be the same (though the actual runtime may be different).

There are a few simple orders of growth. In increasing order they are:

- **log(n)**
- **n**
- **n*log(n)**
- $\boldsymbol{n^2}$
- $\boldsymbol{2^n}$
- **n!**

The higher the order of growth, the slower the program runs- as programmers, we strive for a low order of growth. The order of growth ignores the constant factor in the start of a function. This means that

100N is written as **N**.

We do this because they both scale the same in the long run. Additionally, when calculating the order of growth and two terms are being added or subtracted, we always take the highest term. The result of this is that the order of growth of

NlogN + 1000N + 99 is **NLogN**.

Let's do a basic exercise to see if we understand what is going on. Find the asymptotic running time of the following code where N is the length of the array

```
1 public static void funtimes(int[] a){
2     int j = 0;
3     for(int i = 0; i < 2*a.length ; i++){
4           j ++;
5      }
6 }
```

*The running time of this snippet of code would be **N** because we do a total of **2N steps**. Because in asymptotics we disregard constants, the running time of this snippet.*

3 Asymptotic Notation

We've gone over basic runtimes, but there are various notations used to describe certain needs of the algorithm.

- **Ω** - Big Omega is used to describe the lower bound of a function. For example $n^3 \in \Omega(n^2)$ because the runtime would always be less than n^3
- ***O*** - Big O is used to describe the upper bound of a function. For example $n^2 \in O(n^3)$ because the runtime would always be greater than n^2
- **Θ**- Big Theta is used to describe a tight bound of a function. This means that the upper and lower bound are the same in this scenario. For example $n^2 \in \Theta(n^2)$ because the function always runs in n^2 time

Though technically, Ω and O can be used to be anything lower or higher than the true runtime respectively, we tend to want the tightest bound possible, so we would usually want the tightest possible lowest bound or upper bound (meaning what are the lowest/highest running times that could actually occur).

Instead of saying Ω or O, we could generate a stronger statement by saying Θ in the worst case/best case. This statement is equivalent to what we said in the previous paragraph. It allows us to keep a tight bound and provides us more information than we would get from either Ω or O. The distinction is subtle but let's take this short example.

- In worst case, our code runs in $\Theta(n^2)$ time.
- Our code is bounded by $O(n^2)$

This two statements seem nearly identical, but upon a second glance, you may notice that the first statement tells us that the program can actually run in n^2 time in the worst case (it can never go above) whereas the second statement just says the program will never go above n^2. Basically, the second statement does not really tell us if the program can physically run at that speed. It is imperative that one realizes that omega

Though on first glance, it may seem that Ω means best case and O means worst case, this is not necessarily the situation. In the worst case, you may have a lower bound and upper bound, and a similar situation may occur in the best case. Let's take this pseudocode example

```
1  if(n is even){
2     Randomly pick a number 1 or 2;
3     if(number is 1){
4              do a function that takes n time
5           }
6    else{
7              do a function in n^2
8           }
9  else{
10    Randomly pick a number 1 or 2;
11    if(number is 1){
12             do a function that takes n^3 time
13          }
14   else{
15             do a function in 2^n
16          }
```

In this case, the best case is that n is even; however, there are 2 scenarios that could occur in this situation. The lower bound of the best case is $\Omega(N)$ while the upper bound of the best case is $O(N^2)$. The worst case

is n is odd. The lower bound of the worst case is $\Omega(n^3)$ and the upper bound of the worst case is $O(2^n)$. After this analysis, we can see that Ω and O are not in reference to a specific scenario for a function.

One last thing that we will discuss is *Tilde Notation*. Tilde is a special type of notation denoted with $\sim$. Tilde notation is similar to normal runtime as it looks for the highest running time; however, it takes into account the constant factor of the higher order term. Let's take this example:

$$15n^2 + 100n + 100 \sim 15n^2$$

For function to have the same running time as another function, it just needs to be of the order- $15n^2$ has the same asymptotical running time as $100n^2$. However, to have the same tilde running time, the constant on the highest order terms must be the same so $15n^2$ would only have the same running time as another function with $15n^2$ as it's highest term. Another way of thinking about this is that given two functions $f(x)$ and $g(x)$

$$f(x) \sim g(x) \text{ if } lim_{x\to\infty}\frac{f(x)}{g(x)} = 1$$