**Chapter 06- Big O**

1. **The Big O** is the language and metric used to describe the efficiency of algorithms. Efficiency can be defined as the speed at which an algorithm can perform a task and the amount of system resources that the algorithm uses.
   1. An algorithm that is fast for a small dataset might not be fast for a large dataset.
   2. An algorithm for a less powerful computer system might not be suitable for a computer that is very powerful.
2. **Time Complexity:** Or the runtime of an algorithm in executing a task. The runtime of an algorithm can be modeled in different ways depending on the algorithm. Some algorithms for executing data are controlled by the size of the file where the larger the file the longer the execution time. Others are limited by the transport route of the data, etc…
3. **Space Complexity:** Time is not the only thing that matters in an algorithm. We might also care about the amount of memory or space required by an algorithm.
4. **What is Runtime?** It is the final phase of a computer program’s life cycle, in which the code is being executed on the CPU as machine code. It’s the amount of time it takes for the computer program to finish executing. In general we want the runtime to be as short as possible and to use as few system resources as possible. It’s like tuning a car or a load for a gun cartridge.
5. **When optimizing for runtime, you can usually drop constants.** For example, O(2N) will become O(N). This decreases the lines of code that the algorithm has to execute through each cycle of the algorithm and this can increase the speed. And when an algorithm is being executed millions and billions of times a second, this tweak of dropping a constant matters.
6. **Drop the Non-Dominant Terms**: What do you do about an expression such as O(N2 + N)? N is considered a non-dominant term. It isn’t exactly a constant, but it’s not necessarily important either. As instructions are executed millions and billions of times, the rate of increase of the N2 term quickly overshadows the N term to the point where we can just drop the N term to simplify the algorithm.
7. **Multi-Part Algorithms: Add vs Multiply:** Suppose you have an algorithm that has two steps. When do you multiply the runtimes and when do you add them?
   1. If your algorithm is in the form “do this, then, when you’re done, do that” then you add the run times.
   2. If your algorithm is in the form “do this for each time you do that,” then you multiply the runtimes.
8. **Amortized Time:** amortized time means that when something costly happens, it happens so infrequently, that the cost of this event can be spread over a period of time. For example a medical bill hopefully only happens once in a while, but the cost of that bill can be amortized by paying the bill off in installments over a long period of time, or by paying insurance beforehand which is a form of amortization.
9. **Log N Runtimes:** We commonly see O(log N) in runtimes. Where does this come from?
   1. Let’s look at the example of a **binary search**. In the binary search we are looking for an example x in an N-element **Sorted** array. We first compare x to the midpoint m of the N-Element Sorted array. If x is larger then m, we move to the right section of the array, and if x < m, we move to left section of the sorted array. Then we move to the midpoint of this subsection of the array and keep repeating this process until we get to the number.
   2. So when you see a problem where the number of elements in the problem space gets halved each time, this will likely be a O(log N) runtime.
   3. Finding an element in a **balanced binary search tree** is also a O(log N) runtime.
10. **Recursive Runtimes:** pg 57