**NOTE: All of these can be explained by saying that O(f(n)). Where it measured the worse case scenario always.**

1. **Constant Time - O(1**): This is the best case scenario for your program. When you have n inputs, this will always stay at a constant time complexity of 1.

**Code ex:** This is O(1) since HashMap put and get operations are O(1) time. They get accessed instantly which makes this one of the most used data structures out there.

**A screen shot of a computer code

Description automatically generated**

1. **Logarithmic Time - O( log n):** This would be the second best. This is because the derivative (slope) of 1/(n \*ln10) which means that the slope flattens as n goes towards infinity. This is usually best suited for problems that have a lot of input because of this.

**Code ex:** This is a binary search algorithm. This is O(log n) since you are searching a divided part of the set each time you loop through (in this case half since lp + (rp – lp)/2 will end up being the middle each time. I can explain mathematically, k is an integer (number of iterations/steps), so as you iterate, you are just dividing by 2 repeatedly (n/(2 ^ k) is what we get from this). You can iterate a maximum of 1 time since if the middle is the value, it is only once. n/2^k >= 1 => 2^k >= n => k >= log2(n). Which is log(n).

**A screen shot of a computer program

Description automatically generated**

1. **Linear time - O(n):** This is the time complexity that steadily increased with the slope of 1 (deriving f(n) = n). Given n inputs, the time complexity/operations will equal the number of n inputs. This time complexity is also pretty good.

**Code ex:** Linear search is O(n). This is an example of a linear searching algorithm to find the max element in an array. Since the for loop will iterate n amount of times 1 <= i < arr.length = n. This means that this algorithm has n amount of iterations and thus is O(n).

**A screen shot of a computer program

Description automatically generated**

1. **Quadratic Time -O(n2):** This one is not a good one to use generally as the slope of this is 2n (derivative). You are increasing by about double the amount of a linear time algorithm. Given n inputs, your number of operations will be n \* n. Which is bad for bigger input sizes as you don’t want to have it slow down your program.

**Code ex:** This is an example of a bubble sort algorithm. The outer loop will iterate n times, and the inner loop will also iterate n times. Resulting in an n \* n amount of iterations in the worse case scenario. Thus this is O(n^2).

A screen shot of a computer program

Description automatically generated

1. **Quasilinear Time - O(n log n):** This is better than O(n^2) as the slope of this is log n + 1/ln10. Having approximately the same slope (log n) as one of the previous algorithms (O(log n)) means this is a bad time complexity to go for. This is to say as n increases, So given n inputs, the number of operations if n \* log n. This algorithm is usual for tree sorting and found in a lot of sorting algorithms in general.

**Code ex:** This is an example of code that runs for O(n \* log n) with the while loop as I am dividing x by 2 over and over again. Thus just like how I proved it in binary search, this loop will be O(log n). However, since I put a for loop inside of the log(n), the number of iterations is now considered to be n \* log n. Thus making this example run at O(n \* log n).

**A screen shot of a computer code

Description automatically generated**