Big Oh

Anthony Nguyen

Introduction

So far, when you've programmed, you've really only cared about whether or not your program works. Unfortunately in the real world, when your programs are being benchmarked against other programs for efficiency, you want your program to run not only correctly but as quickly as possible. This is why we need to analyse the runtime and space of the algorithm to make sure it is as low as possible.

We don't just put two computers side by side to measure their runtime because that doesn't tell the whole story. It also takes a lot of work to debug/compile the program and then compare runtime. Using a few calculations, we can decide which algorithm is the better one.

A mathematical tool used to measure runtime and space used of an algorithm is **asymptotic notation**, or the **big Oh**. It gives a very basic but surprisingly accurate runtime for a specific algorithm in its **worst case scenario** (i.e. needing to traverse an entire list to find something).

Assumptions Made when Finding Theoretical Runtime

When we measure the runtime of an algorithm, we assume that the algorithm is working correctly and runs **sequentially** (i.e. in the task main the program runs one line at a time). We also assume that there is no compiler optimization (same runtime on all compilers).

Now these things cannot be ignored in actual runtime, and actual runtime depends on a number of things including hardware, compiler, and programming language.

Big Oh Definition

Relating Two Functions

Big oh is meant to be a simple representation of runtime or space based on the amount of data inputted. It acts as an **upper bound** for the runtime. The function for big Oh is:

$$f(n) = O(n) = O(g(n))$$

f(n) is the actual function that determines runtime

O basically states that it is in big Oh notation and is simplified

n is the amount of data inputted in the algorithm

g(n) is the simplified function for runtime without a constant in front. For example, f(n) = O(2n) would simplify to g(n)=O(n).

Where does this hold true? How much data do you need for this to hold true? Well, the big Oh function holds true when n is big enough (we'll call the 'big enough' or 'threshold' n value n_o) and there is a constant **c** (c is greater than 0) so that

$$f(n) <= cg(n)$$

Whenever $n \geq n_o$

From this we can see the big Oh function multiplied by a constant is basically **an upper bound on the runtime's actual function**.

Proving the Function Falls Under Big Oh

Example: Prove that $f(n) = 8n + 128 = O(n^2)$

1. Classify your f function and your g function. The g function is the one inside the big Oh. g(n) = n^2 and f(n) = 8n + 128.
2. To prove that the function falls under big oh, we need to find both some **constant** and its **threshold n value**. The constant can be chosen arbitrarily (or may be given in the question). We will pick c = 1.
3. Assemble the function $f(n) <= cg(n)$
$$8n + 128 <= n^2$$
4. Solve for the factors of n. These factors will be your n_o values. If there are any positive values for n_o then the condition is satisfied.
$$0 <= n^2 + 8n + 128$$
$$0 <= (n - 16)(n + 8)$$

There is a value n_0 that is positive for the constant c=1 which is also positive. Therefore,

$$f(n) = 8n + 128 = O(n^2)$$

Finding the Big Oh Function

Consider an arbitrary polynomial function, say f(n) = $6n^2$ + 3n + 2. At f(1), the function equals 11. The 2 in the function has a relatively large effect on the answer.

If we increase the value of n to something large, such as 100**, the value 2 has a very small impact on the final answer**. Increasing it even further makes **the 3n term have a small impact.** In fact, for very large numbers, the 6 coefficient in front of the n^2 has a low impact as well. This simplification is essentially what the big Oh notation is.

Big Oh simplification for polynomials is when you eliminate all of the lower growth terms (i.e. 3n, 2 in our example) first, and then eliminate the coefficient in front of the largest growth term, to create g(n), which is the simplified function. For the example above, g(n) = n^2, so that means:

$$f(n) = O\big(g(n)\big) = O(n^2)$$

Types of Big Oh

There are different functions that relate the data inputted to the runtime of the program. Some algorithms run efficiently, so the runtime of the program does not increase very much in comparison to the increase of the data input. Other less efficient algorithms will make runtime increase exponentially as more data is added. Below is a small list of big Oh functions and the explanation of the runtime.

O(1)

O(1) runtime basically says that no matter how big the array, data structure, etc., the algorithm will perform the operation at the same runtime. A good example of this is prepending a linked list. It doesn't matter if there are 2 or 200 links in the linked list, the prepend operation starts at the head of the list so there is no traversal involved. Similarly, if there is a tail for the list, the append operation also has O(1) runtime.

O(n)

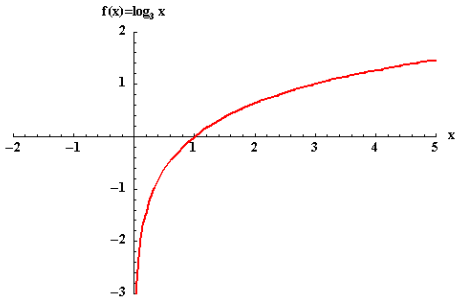
O(n) runtime basically says that the runtime of the operation will be directly proportional to the size of the array, data structure, input, etc. A good example of this is traversing through a linked list. The time it takes to traverse the linked list is directly proportional to the amount of links in the linked list. A linked list with 2 links will be much quicker to traverse than a linked list of 200 links. It also applies to a **for loop**. The difference between the first two conditions of the for loop decide how quickly the operation will be done (i.e. for(i=0, i=n, i++) the runtime will depend directly on n so the big Oh function will be O(n)).

$O(n^k)$

$O(n^k)$ runtime sucks. This is because the runtime of the operation will be proportional to the size of the array to the power of k. In the notes, it'll show up as $O(n^2)$ or $O(n^3)$ or something like that, basically saying the same thing. An example of an algorithm that has $O(n^k)$ runtime is a **nested for loop**. It runs the for loop n times, but within the for loop, there is another for loop that also runs n times. This means the runtime would be proportional to n*n which is n^2. The number of nested for loops determines the **k** value.

O(log(n))

O(log(n)) runtime basically says that if the data inputted increases by a certain amount, the runtime will increase but a whole lot less than that of the data input increase, like the log function. Refer to this graph, where y is runtime and x is data input.



A "Refresher" on Binary Search

An example of O(log(n)) runtime is the **binary search**. If you don't remember, binary search is this:

1. With your sorted, linear array of data, select your upper bound to be the last element in the array and your lower bound to be the first element in the array.
   Example: 1 2 3 4 5 6 7 8 9 10, we are searching for the value 8. Use 1 as the lower bound and 10 as the upper bound.
2. Choose the value directly in the middle of the lower bound and the upper bound. Compare it to the value you are looking for. If it is the value you're looking for, then great, but if not, we move on.
   5 isn't 8 (seriously) so we haven't found our value
3. If the value you are looking for is greater than the middle bound, set the lower bound to the middle bound. If the value you are looking for is lower than the middle bound, set the higher bound to the middle bound.
   8 is greater than 5 so we set the new lower bound to be equal to 5.

4. Repeat steps 2-3 until you have found your value.

The binary search algorithm statistics for runtime are impressive. An array of several hundred thousand values only takes seconds to search in its worst case scenario. This is O(log n) runtime.

O(nlog(n))

N log n is basically a combination of n runtime and log n runtime. This is worse than n and log n runtime on their own. An example of this is running a for loop for a binary search.

Tight Vs Loose Big Oh

Remember that first example I used for proving the function falls under big oh? Notice how the highest exponent of the actual function is 1, while big oh is $O(n^2)$. Remember that big Oh is an upper bound, so even $O(n^3)$ would even hold true. When the big Oh function is polynomial and there is a 'lower' function that can still act as an upper bound for the runtime, this is known as **loose big Oh**.

Tight big Oh is when the highest exponent of the original function is the same as the exponent of what's inside the big Oh function. Refer back to the "finding the Big Oh function" section for an example of big Oh.

Other Asymptotic Notations

Big Omega: Similar to big Oh but it is a lower bound for the runtime function instead of an upper bound

Big Theta: A function that can act as an upper bound AND a lower bound for the runtime function if given different coefficients.