Asymptotic Notations

What are they?

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate. Does the algorithm suddenly become incredibly slow when the input size grows? Does it mostly maintain its quick run time as the input size increases? Asymptotic Notation gives us the ability to answer these questions.

Are there alternatives to answering these questions?

One way would be to count the number of primitive operations at different input sizes. Though this is a valid solution, the amount of work this takes for even simple algorithms does not justify its use.

Another way is to physically measure the amount of time an algorithm takes to complete given different input sizes. However, the accuracy and relativity (times obtained would only be relative to the machine they were computed on) of this method is bound to environmental variables such as computer hardware specifications, processing power, etc.

Types of Asymptotic Notation

In the first section of this doc we described how an Asymptotic Notation identifies the behavior of an algorithm as the input size changes. Let us imagine an algorithm as a function f, n as the input size, and f(n) being the running time. So for a given algorithm f, with input size n you get some resultant run time f(n). This results in a graph where the Y axis is the runtime, X axis is the input size, and plot points are the resultants of the amount of time for a given input size.

You can label a function, or algorithm, with an Asymptotic Notation in many different ways. Some examples are, you can describe an algorithm by its best case, worse case, or equivalent case. The most common is to analyze an algorithm by its worst case. You typically don't evaluate by best case because those conditions aren't what you're planning for. A very good example of this is sorting algorithms; specifically, adding elements to a tree structure. Best case for most algorithms could be as low as a single operation. However, in most cases, the element you're adding will need to be sorted appropriately through the tree, which could mean examining an entire branch. This is the worst case, and this is what we plan for.

Types of functions, limits, and simplification

```
Logarithmic Function - log n
Linear Function - an + b
Quadratic Function - an^2 + bn + c
Polynomial Function - an^z + . . . + an^2 + a*n^1 + a*n^0, where z is some constant
Exponential Function - a^n, where a is some constant
```

These are some basic function growth classifications used in various notations. The list starts at the slowest growing function (logarithmic, fastest execution time) and goes on to the fastest growing (exponential, slowest execution time). Notice that as 'n', or the input, increases in each of those functions, the result clearly increases much quicker in quadratic, polynomial, and exponential, compared to logarithmic and linear.

One extremely important note is that for the notations about to be discussed you should do your best to use simplest terms. This means to disregard constants, and lower order terms, because as the input size (or n in our f(n) example) increases to infinity (mathematical limits), the lower order terms and constants are of little to no importance. That being said, if you have constants that are 2ˆ9001, or some other ridiculous, unimaginable amount, realize that simplifying will skew your notation accuracy.

Since we want simplest form, lets modify our table a bit…

```
Logarithmic - log n
Linear - n
Quadratic - n^2
Polynomial - n^z, where z is some constant
Exponential - a^n, where a is some constant
```

Big-O

Big-O, commonly written as O, is an Asymptotic Notation for the worst case, or ceiling of growth for a given function. Say `f(n)` is your algorithm runtime, and `g(n)` is an arbitrary time complexity you are trying to relate to your algorithm. `f(n)` is O(g(n)), if for any real constant c ($c > 0$), `f(n) <= c g(n)` for every input size n ($n > 0$).

Example 1

```
f(n) = 3log n + 100
g(n) = log n
```

Is `f(n)` O(g(n))? Is `3 log n + 100` O(log n)? Let's look to the definition of Big-O.

```
3log n + 100 <= c * log n
```

Is there some constant c that satisfies this for all n?

```
3log n + 100 <= 150 * log n, n > 2 (undefined at n = 1)
```

Yes! The definition of Big-O has been met therefore `f(n)` is O(g(n)).

Example 2

```
f(n) = 3*n^2
g(n) = n
```

Is `f(n)` O(g(n))? Is `3 * n^2` O(n)? Let's look at the definition of Big-O.

```
3 * n^2 <= c * n
```

Is there some constant c that satisfies this for all n? No, there isn't. `f(n)` is NOT O(g(n)).

Big-Omega

Big-Omega, commonly written as Ω, is an Asymptotic Notation for the best case, or a floor growth rate for a given function.

`f(n)` is Ω(g(n)), if for any real constant c ($c > 0$), `f(n)` is `>= c g(n)` for every input size n ($n > 0$).

Feel free to head over to additional resources for examples on this. Big-O is the primary notation used for general algorithm time complexity.

Ending Notes

It's hard to keep this kind of topic short, and you should definitely go through the books and online resources listed. They go into much greater depth with definitions and examples. More where x='Algorithms & Data Structures' is on its way; we'll have a doc up on analyzing actual code examples soon.

Books

- Algorithms
- Algorithm Design

Online Resources

- MIT
- KhanAcademy