**Complexity and Big-O Notation**

* **Big-O notation** is a specialized notation to measure the complexity of an algorithm.
* We use this notation to measure the relative **runtime** of an algorithm.
* Big-O notation expresses the runtime of an algorithm as a function of:

***a given input of size*** ***(n)***.

* The runtime of an algorithm is proportional to the number of "basic operations" that it performs.
* The total number of basic operations we are given in an algorithm is of size n.

**Big O, Big Theta, and Big Omega**

* Academics use big O, big omega Ω, and big theta ϴ to describe runtimes.

1. O (big O): describes an **upper bound** on the time.

An algorithm that prints all the values in an array could be described as O(N), but it could also be described as O(N2), O(N3), or O(2N) (or many other big O times).

The algorithm is at least as fast as each of these; therefore, they are upper bounds on the runtime.

This is similar to a less-than-or-equal-to relationship.

If Bob is X years old (I'll assume no one lives past age 130), then you could say

X <= $130

X <= $1, 000

X <= $1,000,000

It's technically true (although not terribly useful).

Likewise, a simple algorithm to print the values in an array is O(N) as well as O(N3) or any runtime bigger than O(N).

1. Ω (big omega): describes a **lower bound** to describe runtimes.

Printing the values in an array is O(N) as well as O(logN) and O(1).

After all, you know that it won't be faster than those runtimes.

1. ϴ (big theta): is the **bound** between O and Ω.

That is, an algorithm is ϴ(N) if it is both O(N) and Ω(N).

ϴ gives a **tight bound on runtime**.

* In industry (and therefore in interviews), people seem to have merged ϴ and O together.
* Industry's meaning of big O is closer to what academics mean by ϴ, in that it would be seen as incorrect to describe printing an array as O(N2).
* Industry would just say this is O(N).

**Drop the Constants and Non-Dominant Terms**

* We measure runtime in proportion to the input data size, ***N***.
* **Growth rate:** Change in runtime as ***N*** changes.
* Say an algorithm runs 0.4N3 + 25N2 + 8N + 17 statements.
* Consider the runtime when N is extremely large. (Almost any algorithm is fine if N is small.)
* We ignore constants like 25 because they are tiny next to N.
* The highest-order term (N3) dominates the overall runtime.
* We say that this algorithm runs "on the order of" N3 or O(N3) for short ("Big-Oh of N cubed")
* Big-O notation hides factors with smaller exponents, such as constant factors.
* In particular, it doesn’t matter how long an algorithm takes.
* Any two linear algorithms are considered equally acceptable by this measure.
* There may even be some situations in which the constant is so huge in a linear algorithm that even an exponential algorithm with a small constant would be preferable in practice.

**Growth of Functions**

Graphical user interface, text, application

Description automatically generated

**Remember to “think big” when you consider complexity.**

* The following table lists all the categories of complexity with a certain number of elements to give you a feel of how fast the runtime grows with respect to the number of elements.
* As you can see, **with a small number of elements, the running times don’t differ much.**
* Here, constant factors that are hidden by Big-O notation may have a big influence.
* However, the more elements you have, the bigger the differences in the running times, so constant factors become meaningless.

**Amortized**

* **Amortized** runtime means that a single operation in the long term may take longer than specified.
* An ArrayList is essentially a dynamically resizing array.
* It allows you to have the benefits of an array while offering flexibility in size.
* If you want to insert a new element into an ArrayList, the runtime depends on whether the array has enough memory for one more element.
* If the array hits capacity, the Arraylist class will create a new array with double the capacity and copy all the elements over to the new array.
* How do you describe the runtime of insertion? This is a tricky question.
* There are two different scenarios here:

1. The array could be full.

If the array contains N elements, then inserting a new element will take O(N) time.

You will have to create a new array of size 2N and then copy N elements over.

This insertion will take O(N) time.

1. The array is not full, and you can insert as normal.

We know that the array does not get full very often.

The vast majority of the time insertion will be in O(l) time.

* We need a concept that takes both into account.
* This is what amortized time does.
* It allows us to describe that, yes, this worst case happens every once in a while.
* But once it happens, it won't happen again for so long that the cost is "amortized:'
* In this case, what is the amortized time?
* As we insert elements, we double the capacity when the size of the array is a power of 2.
* So after X elements, we double the capacity at array sizes 1, 2, 4, 8, 16, ..., X.
* That doubling takes, respectively, 1, 2, 4, 8, 16, 32, 64, ..., X copies.
* What is the sum of 1 + 2 + 4 + 8 + 16 + ... + X?
* If you read this sum left to right, it starts with 1 and doubles until it gets to X.
* If you read right to left, it starts with X and halves until it gets to 1.
* What then is the sum of X+ X+ X+ X+...+1?
* This is roughly 2X.
* Therefore, X insertions take 0(2X) time.
* The amortized time for each insertion is 0(1).