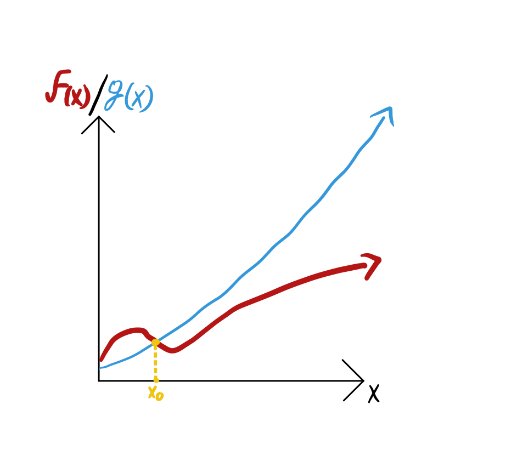
Asymptotic notation

In order to be able to classify the amount of resources an algorithm requires, we must first learn asymptotic notation, which is the standard in computer science to quantify them. Asymptotic notation describes a function in terms of their behavior as the input approaches infinity or: f(x) as x -> infinity. For example let’s start by defining one of the most used notations, the Big O notation.

Say you have a function f(x) which you are comparing to another function g(x). Now let’s assume that we ignore anything that happens to both functions before some arbitrary point x\_0, but after this point, f(x) is always below g(x). See the figure below for an example



If this condition is met, we say that f(x) is Big O of g(x). To be more specific

If for all x s.t. x > x0, g(x) > f(x) -> f(n) belongs O(g(x))

This measure sets an upper bound for our function, it can never be higher than g(x). As I mentioned above, the goal of this formalism is to be transferred to a measure of computing resources. The function f(x) represents the amount of resources needed by our algorithm after taking x bits as input. For example, a common resource to be quantified through this procedure is time. So one could replicate such graph from the figure above, where the y axis represents the number of seconds an algorithm took to finish, and the x-axis the number of bits given to the algorithm as input. Because we always want to minimize the resources used, a higher value in the graph, is a worse value. Due to this, the sentence above “f(x) can never be higher than g(x)” is often written as “f(x) can never be worse than g(x)”. In fact, the Big O of a function, is usually called the “worst case complexity”.

Similarly, there is another notation used called Big Omega which is exactly the opposite of Big O. A function f(x) is said to be Big Sigma of g(x) if after some point x\_0 it is always above g(x)

If for all x s.t. x > x0, g(x) < f(x) -> f(n) belongs sigma(g(x))

This g(x) is then considered the best case complexity of f(x).

Now you are able to quantify resources like a real theoretical computer scientist!

Classical Complexity Theory

The idea behind these blog posts is to stay on the high level ideas in order to make them accessible to larger groups of people. But I must take quick detour to define a language(https://en.wikipedia.org/wiki/Formal\_language) (in the context of mathematics). Strap onto your boots if you are scared of set theory, although I consider languages one of the easiest concepts from the subject, you will be ok I promise.

A language is a set of words built from a defined alphabet. In computer science we usually choose our alphabet to be {0,1}. Therefore, a language can be defined as a collection of bitstrings. For example; one could define a language L as a language where every string must end with a 0

L = {0, 00, 000, …, 10, 100, 1000, …, 110, 1100, …}

So we have languages, and the elements that make up that language, words. Imagine we now build an algorithm M that when you give it a bitstring, it outputs “yes”, if the input is a word of the language, and outputs “no” if the input is not a word that belongs to the language. Then we say that our algorithm M “decides” the language L.

Diagram

Description automatically generated

So far this has all been very abstract, so let’s come down from our theory