Introduction

Big O can be explained as "the growth rate of an algorithm relative to its data structure." To explain the entirety of Big O actually involve explaining some relatively intuitive things so bear with the slightly patronising description of very simple ideas. Take any of the terms used in this document with some salt, I personally did not know them and made some of them up to best describe them.

Funnily, a lot of the concepts are so intuitive and basic that I wrote a tl;dr for some of them to cover the most important points.

I am assuming that whoever reads this have no prior knowledge of any algorithms. Computer scientists would easily be familiar with these algorithms so if that is you, feel free to skip forward

---

Big O Notation

Discrete Algorithms

So suppose we wanted to add 2 numbers, A and B. This operation, computationally takes only 1 step: Adding. Hence you would call this operation O(1). Any operation that is a simple mapping 1 basic arithmetic or logical operation can simply be O(1). Combinations of these operation will then result in the addition of their Big O value.

So if we were to do this: A + B - C. Since this algorithm is a combination of 2 O(1) operation, the resulting Big O is O(2).

(This was almost too trivial to explain by I do it regardless for the sake of completion)

```
TL;DR: Do 1 thing, O(1), do 2 things, O(2)
```

Size dependent algorithms

Certain algorithms are dependent on the scale of the data being processed. Suppose we have a list of 10 numbers, and we wanted to find the smallest number. This would involve scanning the entire list from the beginning to the end while keeping track. So in this case, it would involve scanning through all 10 numbers in the list, a O(10) algorithm. However, suppose we have a list of 27 numbers. The solution will then change to scanning through all 27 numbers, a O(27) algorithm.

As you see, this algorithm of scanning through the list is no longer of fixed size and now dependent on the size of the algorithm. You can see that our algorithm will succeed irrespective to the size of the list. Due to the fact that the algorithm succeeds irrespective to the size but efficiency is dependent onn the size, we can generalise the operation to to consider the size of the data structure, which in this case is the size of the list. So let's say instead of a number, we give the size a generalised variable *n* which acts as a placeholder for list of all size.

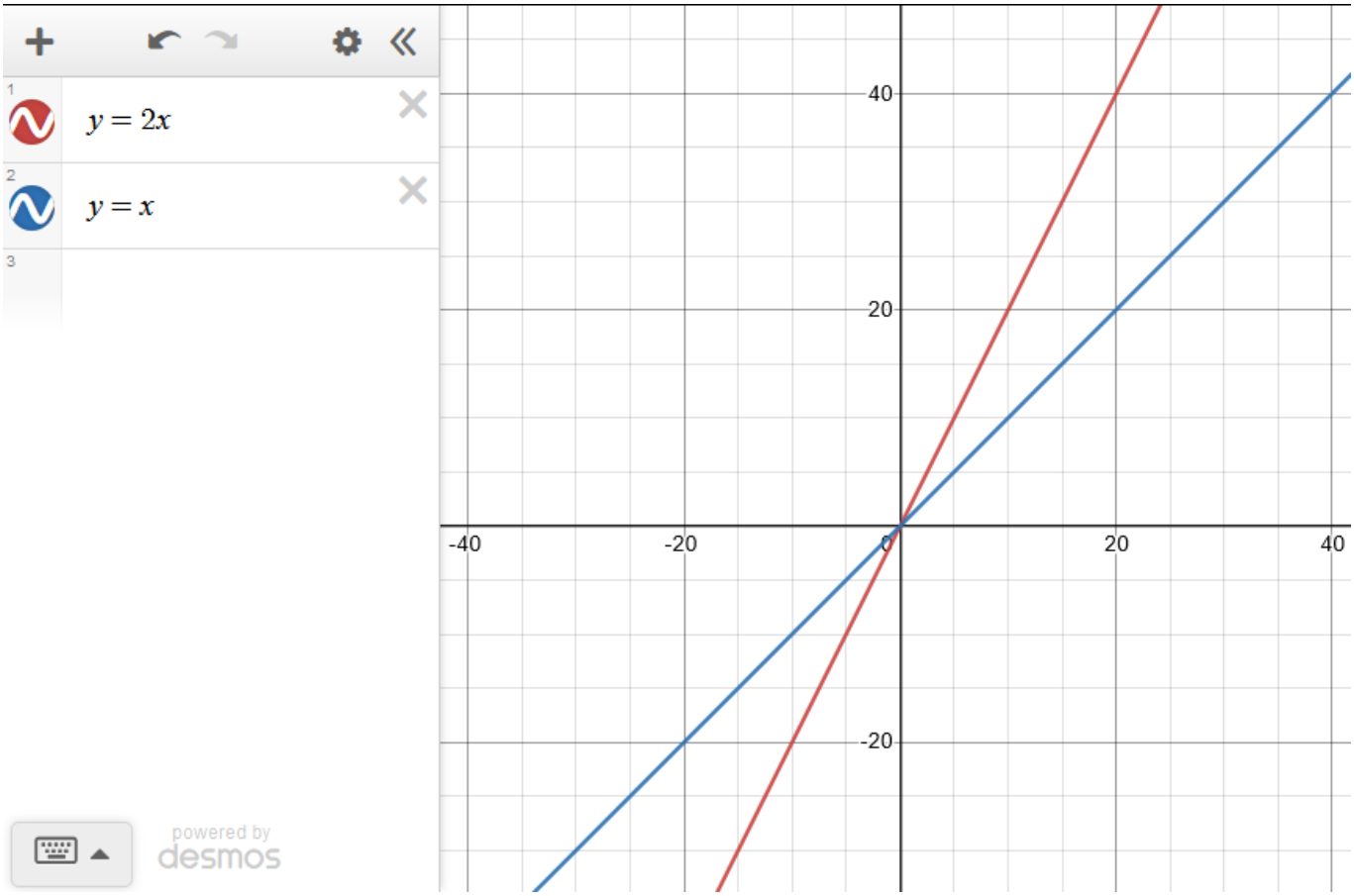
The term "size dependent algorithm" is actually something I made up but it because its the best description of saying: "Algorithm speed depends on size"

```
TL;DR: Do something n times, O(n)
```

Simplification

Now let's look at an algorithm where the complexity is O(2n), which means that the a set number of items needs to be processed twice to get a solution. Let's compare this to another algorithm that solve the same problem in O(n). Now which of the two is faster?

Well intuition dictates that O(n) is surely faster than O(2n) since you are basically operating at half the rate. This is where Big O gets a bit confusing, remember that Big O is the growth rate of an algorithm not necessarily the number of operations the algortihm carries out. When graphically represented, the growth rate of the two algorithm is still linearly increasing, as seen below:



Because their growth rate is straight we can actually simplify the O(c*n) where c is a constant irrespective of the situation, to O(n). Same goes for O(n + c), since the rate of growth is still linear, this too can be simplified to O(n).

However, if you had a variable which is can change within your algorithm (such as 2 list of different size), such as m, then this simplification does not apply. O(mn) and O(m+n) in this case are the simplified case.

```
TL;DR: Constants in Big O is irrelevant given a sufficiently high n
```

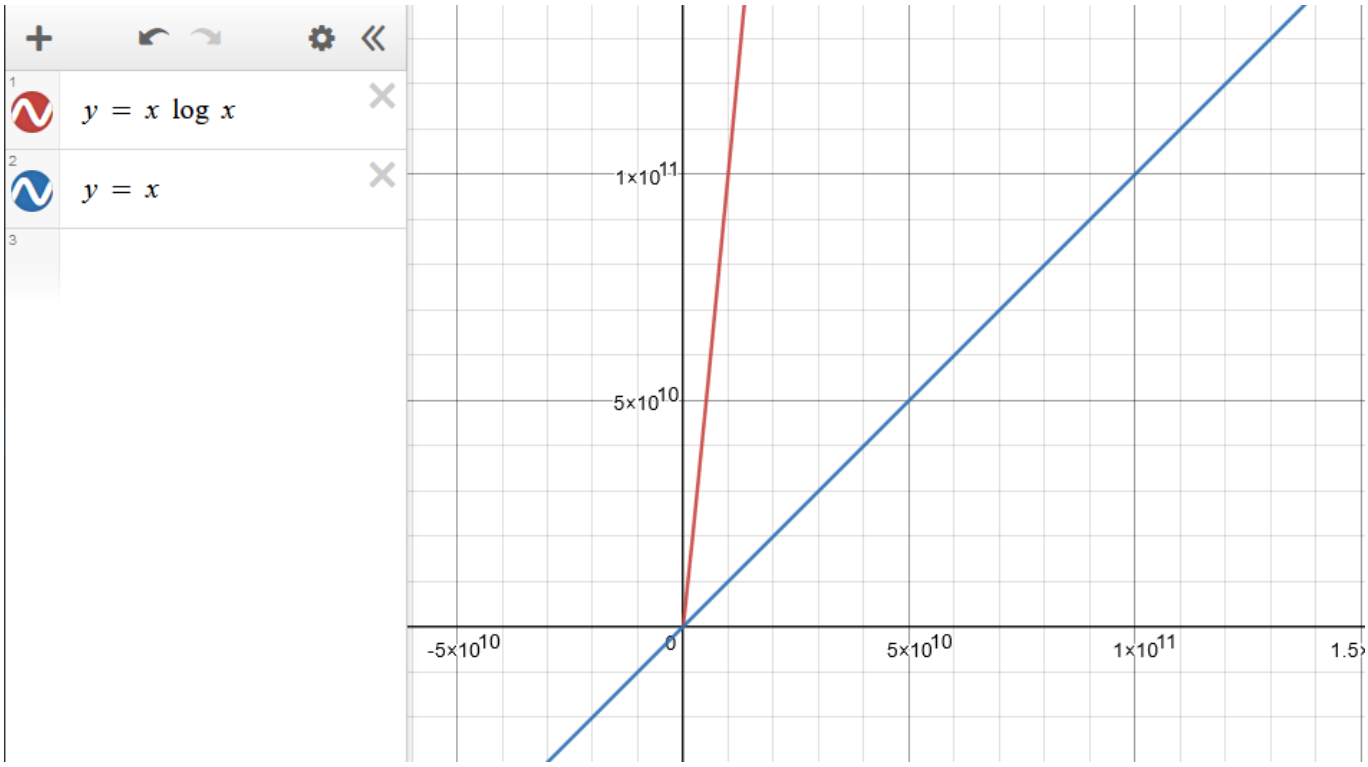
Asymptotic comparison

Let's say you have a solution for a problem. However, within your solution, you have used 2 algorithms with 2 different time complexities: O(n) and O(nlog(n)). Hence, by logic, we can assume that the complexity of this solution is O(n + nlog(n)) right?

Well not exactly. When thinking of time complexity with variable values, you need to think at scale. Remember, the n constant in this case can be any integer, especially any *large* integers.

Suppose we have a huge number, like 1 septillion, as the size of n. If we were to run the said solution with the complexity mentioned, eventually, the O(nlogn) algorithm component will be taking up the majority of the processing time relative to O(n).

Let's graph the two algorithm separately, as seen below:



You can see that, due to the growth rate of the O(nlog(n)), at a sufficiently high n, the difference in the two algorithms process time also increases. Hence O(nlog(n)) will dominate the processing in time. O(n) time complexity slowly becomes irrelevant relative to O(nlog(n))

```
TL;DR: O(n) + O(nlog(n)) = O(nlog(n)). At massive n, O(nlog(n)) takes up the
most time within the algorithm.
```

Known Algorithms

Below are some of known algorithms which would be relevant to divide and conquer. I have included some of the inefficient algorithms as well to better illustrate the faster time complexity

Linear Search `O(n)`

Suppose we have an list of distinct integers of size 10 in no particular order. You find the largest number in the list.

Well the solution in this case is relatively intuitive. You would scan from the top of the list to the bottom of the list, each time keeping track of the largest number encountered and inspecting the current number with the current largest number and replacing accordingly if the current value is greater (I said current a lot)

This is known simply as linear search, where one **linearly** scans through the data structure until the condition has been met. Since this lienar approach requires scanning through every element since you cannot assume that the smallest current number is the smallest without checking every case. The Big O of this algorithm is O(n), *n* being the size of the list.

Binary Search `O(log(n))`

When asked to find a word in the English Dictionary, do you go through letter by letter to find the word? Of course not because that be inefficient. The word "Peanut" clearly comes after the G words but defintely before the Z words, so you will check forwards and backwards over and and over, each time refining your search space until you find "Peanut"

However, the ability to do so in with the knowledge that Dictionary is ordered alphabetically, for if it wasn't you will also have to go through word by word.

Binary search is the mathematically most efficient way to do the search as mentioned. However, it is under the condition that your elements are ordered in some way.

Given a list of n sorted items, binary search starts not at the beginning but in the middle index. If the desired element is by some metric of comparison greater than the middle index, you would eliminate the bottom half of that search space else you would eliminate the top half. You repeat the same process of finding the middle but in your remaining search space, halfing and reducing your search space until you arrive at an index which is equal to your desired element.

Since you are splitting the search space by half, the number of checks required due to limiting the search space is, in the worst case, O(log2(n)). Typically, O(log2(n)) = O(log(n)) since the base 2 is the lowest base of data representation, so unless the specifying another base, log(n) typically refer to base 2.

Insertion Sort `O(n^2)`

Suppose you were given an unordered deck of numbered cards and you were to sort them in increasing order. The most likely method of sorting is going through the unordered deck card by card and placing them in order in a seperate area as you reach.

For example, you started with a 3, then with a 7, you would place the 7 after the 3. Then you get a 5, you will place 5 after 3 but before 7. Then you get a 5, you will place the card either after or before the existing 5 since their order does not matter.

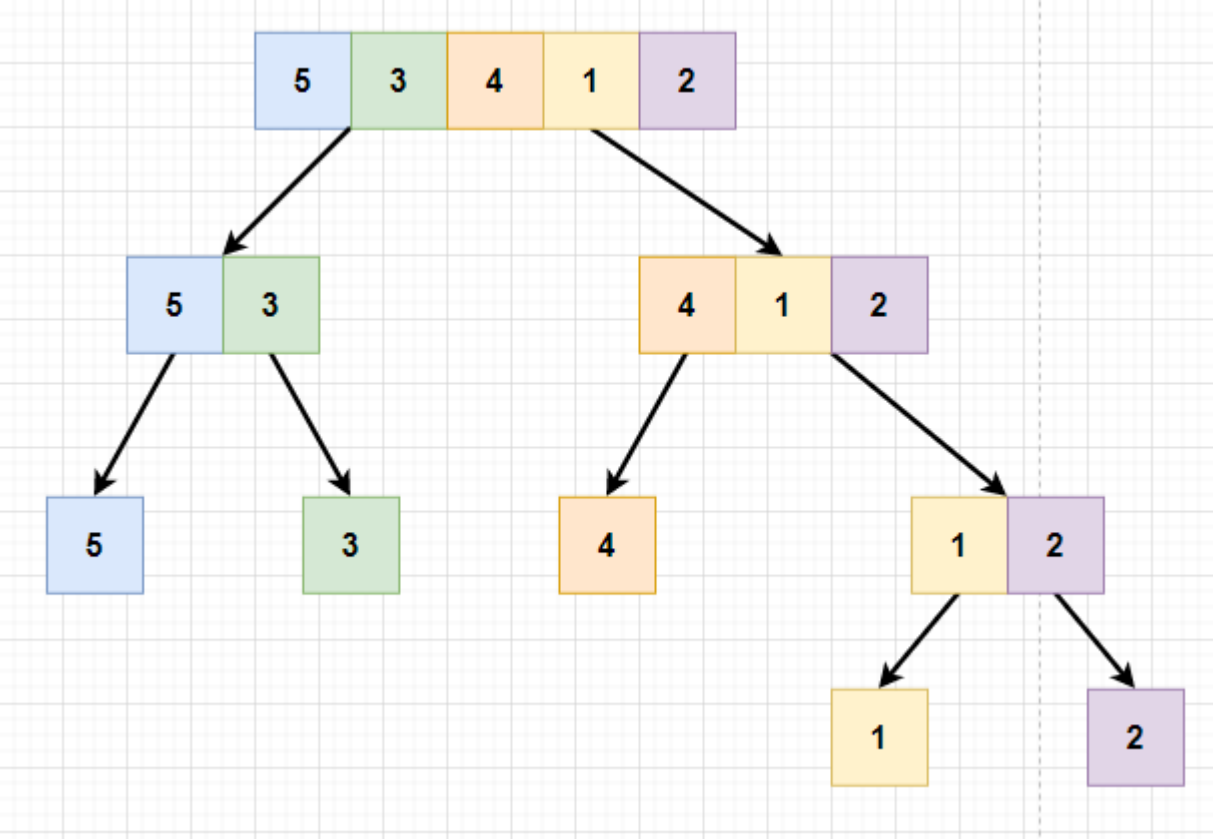
This method is known as insertion sort and in its worst case, you will be given a list in reverse order of what you required ordering. Since you are checking sequentially, you will need to check n cards and place the card at the end of the list which at its largest size n. Hence the time complexity is O(n^2)

Merge Sort `O(nlog(n))`

Merge sort achieve the same as any sorting but it does with the divide and conquer methodology (finally, we reach the content of this topic).

Let's look at Linear for a minute, you notice that, the reasoning for it's O(n^2) time complexity is because the need to scan through the same set of numbers over and over again to find the appropriate index in the current list. However, if we were to split the list into smaller manageable parts

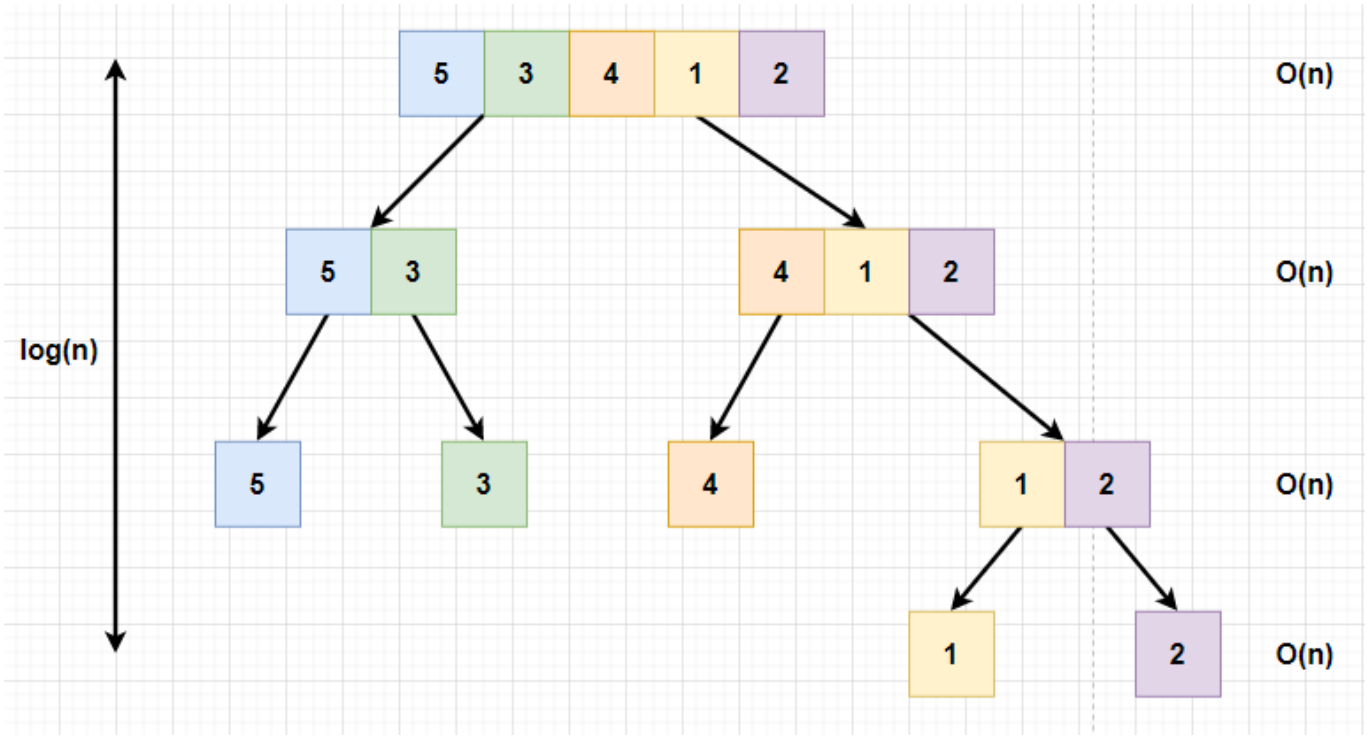
Merge sort work as follows: Given the list, you split the list down the middle into 2 lists. Then you repeat this process with the smaller lists, splitting them in half over and over until you reach a single element. This can be considered as a recursion which means it can be represented as a tree, as seen below:



Once you reach the one element (the leaf node), you recurse back up and sort each chunk. Using our diagram, once you reach 5, since its a single number, no sorting is required, same goes for 3. However, once go up 1 level, you sort in order between 5 and 3, resulting in a set of (3,5).

Once you have 2 sets of (3,5) and (1,2,4), you sort these by placing each element of either set and ploace them in the respective index in the other second, combining them into 1 set.

The depth of the tree is log(n) through splitting the list in half until its atomic element. You operate on every element at each level at worst once, you are operating n elements and hence the time complexity is O(n). See diagram for visual aid:



So since there is log(n) levels and you are operating O(n) at each level. The time complexity is O(nlog(n)).

Average vs Worst Case

Now let's solve a simple problem: Finding a number in a list of size n. Now, as you look for the number, you will be scanning through every element one by one in the list. So with everything we have learned so far, it is safe to say that the time complexity of this algorithm is O(n).

Or is it?

Practically, if I was to hand you a list to scan through, the chances of the number you are looking for being at the very end is very slim. You are more likely to find the number somewhere in the middle of the list. So can we say this algorithm is O(n) if practically, you will find the number in the middle of the list more frequently than at the end

This is where we come to the idea of *Average Case* and *Worst Case*. Practically, a lot of algorithms discussed so far do not actually require the full time required and will complete early. This is what we called the average case, where the algorithm, in practice, is faster.

However, computer scientist are pessimistic, scorned, people. We rather be pleasently surprised that something is faster than be optimistic and have something be late. This would matter especially when dealing with programs that require strict schedules. A computer can take a completed output and use it when it needs to, but if it does not have what it needs when it needs it, it may throw a tantrum that results in either: A progress bar being a tad slower to a plane crash catastrophie.

So this is where *Worst case* comes in. In this case, we assume that every algorithm will have to process every bit of information

So when dealing with algorithm questions, be sure to consider whether the average case scenario is something viable to consider. If they make no mention of this in the specification, be pessimistic and assume Worst Case.

You might be wondering why we should even consider Average Case, when ultimately Worst Case is the safest option? Well, in practice, the likelihood of you always reaching or even coming close to the worst case scenario are so improbable that you might as well not consider it. Again it's really dependent on the circumstance and and how critical it is for the application to terrminate in a given time.

Let's talk about Hashmap for second. Hashmap are an interesting data structure where where given an element, the value will be hashed to find its index in a key. Given the key, it will then map said key to said element. If the element needs to be found, all you need to index via its hashed key. Time complexity is hence O(1)

However, there could be a small chance that the key produced is already existing. What happens is *chaining* where the element is placed next to the existing element. In the future if you index said key, it will then do a linear search on the hash map index until the value its found.

This is where worst case comes in. In the most unbelievable worst case, every element is hashed to same index and placed sequentially as per insertion order. This then time complexity is the same as linear search: O(n).