https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

NOTE appended comments (p. 3)!

A beginner's guide to Big O notation

rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

Anyone who's read Programming Pearls or any other Computer Science books and doesn't have a grounding in Mathematics will have hit a wall when they reached chapters that mention O(N log N) or other seemingly crazy syntax. Hopefully this article will help you gain an understanding of the basics of Big O and Logarithms.

As a programmer first and a mathematician second (or maybe third or fourth) I found the best way to understand Big O thoroughly was to produce some examples in code. So, below are some common orders of growth along with descriptions and examples where possible.

O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
    return elements[0] == null;
}
```

O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the `for` loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(IList<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

O(N^2)

O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N^3), O(N^4) etc.

```
bool ContainsDuplicates(IList<string> elements)
{
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]) return true;
        }
    }

    return false;
}
```

$O(2^N)$

$O(2^N)$ denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an $O(2^N)$ function is exponential - starting off very shallow, then rising meteorically. An example of an $O(2^N)$ function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as **O(log N)**. The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

23 June 2009

==========
https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

> It might be worth mentioning that O(log N) is the theoretical limit for searching a data set.

> O(log N) is log base 2 as well. This is an important distinction.

> Regarding O notation there is no distinction between O(log N), O(ln N) or logarithms of any base. This is because O notation is saying, asymptotically, that the algorithm has a time complexity limited by the log function. A logarithm of any base b is a real multiple of any generic logarithm: log_b N = log N / log b. These are equivalent in O notation because of its definition: “if g(x) is O(f(x)) there exists some real constant M such that g(x) <= M f(x)” obviously if we multiply by 1/log b, there is another real constant N = M / log b that satisfies this definition.

> Nice article, but I'll have to quibble with this: "Big O specifically describes the worst-case scenario". That is not the case. Big-O notation just describes asymptotic bounds, so it is correct to say something like, for example, "Quicksort is in O(n!)," even though Quicksort's actual worst-case running time will never exceed O(n^2). All Big-O is saying is "for an input of size n, there is a value of n after which quicksort will always take less than n! steps to complete." It does not say "Quicksort will take n! steps in the worst case."

==========
https://hn.svelte.technology/item/16635440
https://news.ycombinator.com/item?id=9111362
User-defined Order in SQL

... - walk the list in the database with a recursive common table expression. CTE's aren't appropriate for this; it'll be slow, and we'll run up against a recursion limit for large lists.- walk the list in the database with cursors/loops/etc. Very icky, and breaks composability.

> You are right and make many succinct points. I imagined this in the context of the todo list example given in the article, and assumed there would be a sane limit on items in a list, and that each record in the database would be tied to a list ID, or user ID, or some other method to allow only pulling the items that are actually in the list.

That being said, this would not alleviate the problems you mentioned when walking the list. It would need to be arranged once retrieved either by the client, or by the backend before passing to the client. I imagined that I would traverse the list recursively to order it before passing it off to the client. This should take O(n) time, since we only need to traverse the list once, and we haven't retrieved from the database any unrelated rows (list ID, user ID, etc - there has to be some boundary in place).

==========
https://blog.jooq.org/tag/recursive-sql/

... As with every abstraction, you will still need to know the basics of what’s going on behind the scenes in a database to help the database make the right decisions when you query it. For instance, it makes sense to:

* Establish a formal foreign key relationship between the tables (this tells the database that every address is guaranteed to have a corresponding user)

* Add an index on the search field: The country (this tells the database that specific countries can be found in O(log N) instead of O(N))

VICTORIA:

With a (balanced) B-tree index in Postgres | SQL, the search time| space| ... complexity is O(log n); see (e.g.)
https://en.wikipedia.org/wiki/B-tree

==========
https://news.ycombinator.com/item?id=9111362
Tree structure query with PostgreSQL

... I haven't done nearly enough research to back this point of view, but it seems to me that if you really need to make a recursive query over a data tree, you should maybe reconsider how that tree is being modeled. The closure table model [1] is simple to query non-recursively, and unless your tree is pathological, it doesn't add that mcuh space overhead. I suppose though that a built-in recursive query might not be so bad though, if it's just a whole bunch of indexed row lookups within one query.Perhaps there are situations where recursive queries really are the only solution. I'd love to hear some examples.
http://technobytz.com/closure_table_store_hierarchical_data.html

> How do you determine if your tree is pathological? I.e., what types of tree structures would you find the closure table model to be pathological enough to warrant other modeling options?

>> Good question -- so the most pathological tree of n nodes would be one with depth of O(n). If this was the case, the closure table would be of size O(n^2). In tree applications I can think of--like an org chart or site comments--the depth is O(log(n)), in which case the closure table is just O(n). Note that this same kind of pathological tree would be hell for a recursive query too, just trading space for time.