2-big-o-notation

Contents

Introduction **2**

What is it **2**

Big theta and omega **2**

Common complexities **2**

Best & worst **3**

Space complexity **3**

Drop the constants **3**

Drop non-dominant terms **3**

Add vs Multiply of complexities **4**

Amortized time **4**

Log N run times **4**

Recursive run times **5**

- Introduction
- What is it
- Big theta and omega
- Common complexities
- Best & worst
- Space complexity
- Drop the constants
- Drop non-dominant terms
- Add vs Multiply of complexities
- Amortized time
- Log N run times
- Recursive run times

Introduction

One of the most important concepts to get right, and understand, it is used to describe the efficiency of algorithms. Not understanding it can really hurt you, and might be judged harshly.

What is it

Big O is also called asymptotic run time. In very crude terms, no matter how big the constant is, and how slow the linear increase is, linear will at some point surpass the constant progression.

- O(1) - constant complexity, meaning that no matter the input parameters, the run time will always take a given constant amount of time
- O(n) - non-constant complexity, meaning that the complexity of the run time is directly correlated to the input arguments or parameters

There are many types of complexity such as `log(n)` or `n^2`, or `2^n`. Each of which describes non-constant run time complexity. There is a non-fixed list of possible non-constant complexity run times.

The run time could also have a dependency on multiple variables, instead of just n, which represents one of the input parameters, we could have n*m, where m is another input parameter, which is in play. The dependent arguments might also be indirectly connected to the input, and be a function of the input instead.

`Big O allows us to describe how the run or space complexity scales, and not to measure absolute units of time or space.`

Big theta and omega

In academia, there is - big O, big theta, and big omega. Three different complexities describing different ranges of run times

- `big O` - in academia that describes an upper bound on the run time. Something prints values of an array of N items, can be described as O(N), but it could also be described as O(N^2), O(N^3), O(N^4). The printing is at least as fast as each of these run times, it will however never be less than, the lowest complexity run time, in the example above that is O(N).
- `big theta` - is the same concept but for a lower bound. Where the inverse is true, meaning that an algorithm would not be slower than than O(N)
- `big omega` - is when big O and big theta, give a tight bound on the run time, meaning when they converge to the same run time complexity. This is what the industry means by big O, and not the academic, big O, which only describes an upper boundary, as mentioned above.

Common complexities

In practice we have the following 7 most commonly occurring complexities, each worse than the other, they are listed in order of the fastest to the slowest

1. Constant - `O(1)` - array access
2. Logarithmic - `O(log(n))` - binary search
3. Linear - `O(N)` - array print
4. N*Logarithmic - `O(N*log(n))` - quick sort
5. Quadratic - `O(N^2)` - bubble sort
6. Exponential - `O(2^N)` - Fibonacci sequence

7. Factorial = `O(n!)` - string permutations

Best & worst

Generally when talking best, worst and expected run times, in the industry we focus on the expected and worst, where usually the expected is the worst, or in other words the worst case is not a case where the algorithm fails totally. There are some cases where the worst case for a given algorithms differs significantly from the expected one.

Take quick sort, based on the pivot element we choose, and the direction of the sorting, descending or ascending, if we always take the first element for a pivot, and we sort in reverse order the array won't be divided in half by the pivot, rather it will be shrunk down only by a single element, making the algorithm effectively of linear run time.

Space complexity

The space complexity is a parallel concept to the run time complexity, meaning that each algorithm, requires a specific amount of memory or space to execute in, based on the type of algorithm, that could be constant, or non-constant space, for example take a regular loop and a recursive print of an array of N elements, the loop variant will take constant space for the entire execution, however it would take `O(N)` space if we use recursion since the call stack, and stack frame function will constantly grow on each call for the next element.

Drop the constants

It is possible for an O(N) to be faster than O(1), since the big O describes the rate of increase, however a constant run time does not mean instant execution. For this reason we drop the constants in run time meaning that the following are equivalent - `O(1*N)== O(2*N)== O(3*N)== O(4*N)== etc`. That is because the constant time no matter how big in absolute units of time or space, will/is always constant, and is not correlated to the input parameters.

Take for example, an algorithm, that finds the max and min element in an array, you could go about this two different ways

1. Two for loops, one for the max and one for the min element - `O(2N)`
2. One loop to find both the max and min element - `O(N)`

Well in either case the complexity here is O(N) for time and O(1) for space, both of these algorithms, scale the same way with N. Having one combined for loop that does both tasks or two different ones does not change the run time complexity.

Drop non-dominant terms

Since we can drop the constants, we could also take into account that some complexities might have multiple terms dependent on the input arguments, where one of them is so much more dominant / bigger than the other could be simply removed, since it will not affect the complexity scaling in any significant way

1. O(N^2 + N) - O(N^2)
2. O(N^2 + 10*N) - O(N^2)
3. O(N^3 + log2(N)) - O(N^3)
4. O(2^N + 1000*N) - O(2^N)

Now this is only relevant when we talk about the same input parameter correlation, we cannot reduce a complexity of two different terms, for example the following `O(A + B)`, cannot be reduced without having some information about the parameters beforehand, if we know that A is significantly more dominant than B, then yes, but that requires additional data or information about our input.

Add vs Multiply of complexities

When would one add two complexities or multiply them. The rules are as follows

- if for `each A chunks of work, we do B chunks of work`, then we multiply, this is mostly expressed as nested loops or similar loop like actions.
- if however the work done for `each chunk of A and each chunk of B,` then we add the run times, this is similar to have loops executed one after the other for example.

Amortized time

What this term describes, is complexities where once in a while, worst case scenario will occur for sure, and we cannot avoid it. Think about a self re sizing array, where once the array is full, the internal implementation doubles the old size into a new array and copies the old elements into the new one. For this algorithm the regular complexity in time is O(1), but, once in a while it will degrade to O(N), which is what amortized time describes.

Log N run times

In most algorithms we usually deal with log with base 2, since the results we have to go through, halve by two each time we visit them. So taking a sorted array, to which we apply binary search, where each iteration we halve the array we have to look through. Taking as an example array with 16 elements, to find the element we are looking for we have to make at most 4 comparisons. The run time still scales with the input, but it is not linear, it is faster, instead of 16 comparisons we do 4.

`log2(n)= k == 2^k = n` - the general log formula, 2 raised to what power would result in n.

`log2(16)= 4 == 2^4 = 16` - the example above will look like this, expressed as a log of base 2 of 16, which is 4.

Why 4, we start with 16 elements, take the mid point, compare and sub-divide into 2, until we have only 1 element, or have found the element being search

0. 16 -> divide by 2
1. 8 -> divide by 2
2. 4 -> divide by 2
3. 2 -> divide by 2
4. 1 -> found target

When we see a problem space where the solution space divides the data by two, it is more often than not a base two log of the input n - `log(n)`

If the problem space was divided by 4 or 8 or 16 times instead on each step, then our log base would be either 4, 8 or 16. For example `log4(16)` will be 2, meaning that we can find solution in 2 steps instead of 4 (see example above)

Note that the base of the log, does not matter, the complexity will still be log of some base of the number of input arguments, could be `log10(n)or log4(10)` etc. The reason it does not matter, is because the curve

described by log, is still the same, no matter the base of the logarithm, remember that big O does not evaluate absolute run time & space complexity, rather it describes scaling

What the log describes is the type of scaling in relation to the input, we are not looking for which log base produces a smaller / better absolute value of steps being executed, instead we are looking at the overall scaling of the input in relation to the solution we have

Recursive run times

Given the following example, we have to find out what is the run-time and space complexity of the provided function

```
int f(int n) {
    if (n <= 1) {
        return 1;
    }
    return f(n - 1) + f(n - 1);
}
```

Calling the above function with f(4), what is the complexity of this function. Well for f(4), the function would call f(3) twice, then f(3), would call f(2) twice, and so forth, so with each level we double the calls, until we reach the bottom or f(1).

The number of calls would then be 2^D (where D is the depth of the recursion tree formed by the call stack). In our case, the depth D is 4, the bottom most level is 3, the level above it is 2, the one above it is 1, and the final one with the root node, where we start f(4) is 0. Therefore at most `2^4 - 1 = 15` function calls. Note that each level (in this specific example, which is a perfectly balanced binary tree) has exactly `2^(current-depth)` number of nodes

```
                   f(4)                          2^0 = 1
             /              \
        f(3)                  f(3)               2^1 = 2
      /      \              /      \
  f(2)      f(2)       f(2)      f(2)            2^2 = 4
  /  \      /   \      /   \     /   \
f(1) f(1) f(1) f(1)  f(1) f(1) f(1) f(1)         2^3 = 8
```

What is important to note here is what N is, it is not a number of elements of an array, or the size of the binary tree, nor is it the depth, what N is, in this case is simply a number, but that number governs our algorithm and how it scales, so the algorithm scales with an absolute number N (an integer, such as 1,2,3...N). To calculate N we do 2 recursive calls to calculate N-1. With each increase in N, the number of calls doubles. The branching factor of our algorithm is 2 (we have 2 function calls per invocation of f)

Note that, by coincidence, the depth (D) of the tree (formed by the call stack) and the input here match, so `N == D`, levels, but do not confuse the depth of the tree, and the input N. The depth does not express the run-time complexity, or in other words the `run-time complexity is not governed by the depth`. The generated call stack tree, and its depth is a bi product of the input and algorithm itself The algorithm scales with the input `number N` and `N` alone.

Changes in `N` and the algorithm will produce different depth, different scaling and different run- time and space complexities (e.g, we had 3 recursive calls to N-1 to calculate N)