Chapter 3: Big-O Notation

Definition

The Big-O notation is at its heart a mathematical notation, used to compare the rate of convergence of functions. Let `n -> f(n)` and `n -> g(n)` be functions defined over the natural numbers. Then we say that `f = O(g)` if and only if `f(n)/g(n)` is bounded when n approaches infinity. In other words, `f = O(g)` if and only if there exists a constant A, such that for all n, `f(n)/g(n) <= A`.

Actually the scope of the Big-O notation is a bit wider in mathematics but for simplicity I have narrowed it to what is used in algorithm complexity analysis : functions defined on the naturals, that have non-zero values, and the case of n growing to infinity.

What does it mean ?

Let's take the case of `f(n) = 100n^2 + 10n + 1` and `g(n) = n^2`. It is quite clear that both of these functions tend to infinity as n tends to infinity. But sometimes knowing the limit is not enough, and we also want to know the *speed* at which the functions approach their limit. Notions like Big-O help compare and classify functions by their speed of convergence.

Let's find out if `f = O(g)` by applying the definition. We have `f(n)/g(n) = 100 + 10/n + 1/n^2`. Since `10/n` is 10 when n is 1 and is decreasing, and since `1/n^2` is 1 when n is 1 and is also decreasing, we have `f(n)/g(n) <= 100 + 10 + 1 = 111`. The definition is satisfied because we have found a bound of `f(n)/g(n)` (111) and so `f = O(g)` (we say that f is a Big-O of `n^2`).

This means that f tends to infinity at approximately the same speed as g. Now this may seem like a strange thing to say, because what we have found is that f is at most 111 times bigger than g, or in other words when g grows by 1, f grows by at most 111. It may seem that growing 111 times faster is not "approximately the same speed". And indeed the Big-O notation is not a very precise way to classify function convergence speed, which is why in mathematics we use the equivalence relationship when we want a precise estimation of speed. But for the purposes of separating algorithms in large speed classes, Big-O is enough. We don't need to separate functions that grow a fixed number of times faster than each other, but only functions that grow *infinitely* faster than each other. For instance if we take `h(n) = n^2*log(n)`, we see that `h(n)/g(n) = log(n)` which tends to infinity with n so h is *not* O(n^2), because h grows *infinitely* faster than n^2.

Now I need to make a side note : you might have noticed that if `f = O(g)` and `g = O(h)`, then `f = O(h)`. For instance in our case, we have `f = O(n^3)`, and `f = O(n^4)`... In algorithm complexity analysis, we frequently say `f = O(g)` to mean that `f = O(g)` *and* `g = O(f)`, which can be understood as "g is the smallest Big-O for f". In mathematics we say that such functions are Big-Thetas of each other.

How is it used ?

When comparing algorithm performance, we are interested in the number of operations that an algorithm performs. This is called *time complexity*. In this model, we consider that each basic operation (addition, multiplication, comparison, assignment, etc.) takes a fixed amount of time, and we count the number of such operations. We can usually express this number as a function of the size of the input, which we call n. And sadly, this number usually grows to infinity with n (if it doesn't, we say that the algorithm is O(1)). We separate our algorithms in big speed classes defined by Big-O : when we speak about a "O(n^2) algorithm", we mean that the number of operations it performs, expressed as a function of n, is a O(n^2). This says that our algorithm is approximately as fast as an algorithm that would do a number of operations equal to the square of the size of its input, *or faster*. The "or faster" part is there because I used Big-O instead of Big-Theta, but usually people will say Big-O to mean Big-Theta.

When counting operations, we usually consider the worst case: for instance if we have a loop that can run at most n times and that contains 5 operations, the number of operations we count is 5n. It is also possible to consider the average case complexity.

Quick note : a fast algorithm is one that performs few operations, so if the number of operations grows to infinity *faster*, then the algorithm is *slower*: O(n) is better than O(n^2).

We are also sometimes interested in the *space complexity* of our algorithm. For this we consider the number of bytes in memory occupied by the algorithm as a function of the size of the input, and use Big-O the same way.

Section 3.1: A Simple Loop

The following function finds the maximal element in an array:

```
int find_max(const int *array, size_t len) {
    int max = INT_MIN;
    for (size_t i = 0; i < len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

The input size is the size of the array, which I called `len` in the code.

Let's count the operations.

```
int max = INT_MIN;
size_t i = 0;
```

These two assignments are done only once, so that's 2 operations. The operations that are looped are:

```
if (max < array[i])
i++;
max = array[i]
```

Since there are 3 operations in the loop, and the loop is done n times, we add `3n` to our already existing 2 operations to get `3n + 2`. So our function takes `3n + 2` operations to find the max (its complexity is `3n + 2`). This is a polynomial where the fastest growing term is a factor of n, so it is O(n).

You probably have noticed that "operation" is not very well defined. For instance I said that `if (max < array[i])` was one operation, but depending on the architecture this statement can compile to for instance three instructions : one memory read, one comparison and one branch. I have also considered all operations as the same, even though for instance the memory operations will be slower than the others, and their performance will vary wildly due for instance to cache effects. I also have completely ignored the return statement, the fact that a frame will be created for the function, etc. In the end it doesn't matter to complexity analysis, because whatever way I choose to count operations, it will only change the coefficient of the n factor and the constant, so the result will still be O(n). Complexity shows how the algorithm scales with the size of the input, but it isn't the only aspect of performance!

Section 3.2: A Nested Loop

The following function checks if an array has any duplicates by taking each element, then iterating over the whole array to see if the element is there

```
_Bool contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len; j++) {
            if (i != j && array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

The inner loop performs at each iteration a number of operations that is constant with `n`. The outer loop also does a few constant operations, and runs the inner loop `n` times. The outer loop itself is run `n` times. So the operations inside the inner loop are run `n^2` times, the operations in the outer loop are run `n` times, and the assignment to `i` is done one time. Thus, the complexity will be something like `an^2 + bn + c`, and since the highest term is `n^2`, the O notation is `O(n^2)`.

As you may have noticed, we can improve the algorithm by avoiding doing the same comparisons multiple times. We can start from `i + 1` in the inner loop, because all elements before it will already have been checked against all array elements, including the one at index `i + 1`. This allows us to drop the `i == j` check.

```
_Bool faster_contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            if (array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

Obviously, this second version does less operations and so is more efficient. How does that translate to Big-O notation? Well, now the inner loop body is run `1 + 2 + ... + n - 1 = n(n-1)/2` times. This is *still* a polynomial of the second degree, and so is still only `O(n^2)`. We have clearly lowered the complexity, since we roughly divided by 2 the number of operations that we are doing, but we are still in the same complexity *class* as defined by Big-O. In order to lower the complexity to a lower class we would need to divide the number of operations by something that *tends to infinity* with `n`.

Section 3.3: O(log n) types of Algorithms

Let's say we have a problem of size n. Now for each step of our algorithm(which we need write), our original problem becomes half of its previous size(n/2).

So at each step, our problem becomes half.

Step	Problem
1	n/2
2	n/4
3	n/8
4	n/16

When the problem space is reduced(i.e solved completely), it cannot be reduced any further(n becomes equal to 1) after exiting check condition.

1. Let's say at kth step or number of operations:

   problem-size = 1

2. But we know at kth step, our problem-size should be:

   problem-size = n/2k

3. From 1 and 2:

   n/2k = 1 or

   n = 2k

4. Take log on both sides

   loge n = k loge2

   or

   k = loge n / loge 2

5. Using formula **logx m / logx n = logn m**

   k = log2 n

   or simply k = log n

> Now we know that our algorithm can run maximum up to log n, hence time complexity comes as O(log n)

A very simple example in code to support above text is :

```
for(int i=1; i<=n; i=i*2)
{
    // perform some operation
}
```

So now if some one asks you if n is 256 how many steps that loop(or any other algorithm that cuts down it's problem size into half) will run you can very easily calculate.

k = log2 256

k = log2 2 8 (=> logaa = 1)

k = 8

Another very good example for similar case is **Binary Search Algorithm**.

```
int bSearch(int arr[],int size,int item){
 int low=0;
 int high=size-1;

 while(low<=high){
     mid=low+(high-low)/2;
     if(arr[mid]==item)
         return mid;
     else if(arr[mid]<item)
         low=mid+1;
     else  high=mid-1;
     }
  return –1;// Unsuccessful result
}
```

Section 3.4: An O(log n) example

Introduction

Consider the following problem:

`L` is a sorted list containing `n` signed integers (`n` being big enough), for example `[-5, -2, -1, 0, 1, 2, 4]` (here, `n` has a value of 7). If `L` is known to contain the integer 0, how can you find the index of 0 ?

Naïve approach

The first thing that comes to mind is to just read every index until 0 is found. In the worst case, the number of operations is `n`, so the complexity is O(n).

This works fine for small values of `n`, but is there a more efficient way ?

Dichotomy

Consider the following algorithm (Python3):

```
a = 0
b = n-1
while True:
  h = (a+b)//2 ## // is the integer division, so h is an integer
  if L[h] == 0:
    return h
  elif L[h] > 0:
    b = h
  elif L[h] < 0:
    a = h
```

`a` and `b` are the indexes between which 0 is to be found. Each time we enter the loop, we use an index between `a` and `b` and use it to narrow the area to be searched.

In the worst case, we have to wait until `a` and `b` are equal. But how many operations does that take? Not n, because each time we enter the loop, we divide the distance between `a` and `b` by about two. Rather, the complexity is O(log n).

Explanation

Note: When we write "log", we mean the binary logarithm, or log base 2 (which we will write "log_2"). As O(log_2 n) = O(log n) (you can do the math) we will use "log" instead of "log_2".

Let's call x the number of operations: we know that 1 = n / (2^x).

So 2^x = n, then x = log n

Conclusion

When faced with successive divisions (be it by two or by any number), remember that the complexity is logarithmic.