Chapter 1

Basic Analysis of Algorithms

Let's understand how to analyse an algorithm with respect to:-

1. Execution Time
2. Memory Consumed

Big-O Analysis of Algorithms

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time. Big-O notation tells you how efficient the algorithm is.

Examples

1. Linear Time

```
const linear_time=n=>{
   for(let i=0;i<n;i++){
       console.log(i)
   }
}
linear_time(5)


//Output
0
1
2
3
4
5
```

2. Quadratic Time

```
const quadratic_time=n=>{
   for(let i=0;i<n;i++){
       console.log(i)
       for(let j=i;j<n;j++){
           console.log(j)
       }
   }
}
quadratic_time(5)
//Output
0
1
2
3
4
5
```

Rules of Big-O Notation:

Let's represent an algorithm's complexity as $f(n)$, n represents the number of inputs, $f(n)$time represents the time needed, and $f(n)$ space represents the space (additional memory) needed for the algorithm. It can be challenging to calculate $f(n)$. But Big-O notation provides some fundamental rules that help developers compute for $f(n)$.

- **Coefficient rule**: - If $f(n)$ is $O(g(n))$, then $k(f(n))$ is $O(g(n))$, for any constant k > 0.
- **Sum rule** : If $kf(n)$ is $O(h(n))$ and $g(n)$ is $O(p(n))$, then $f(n) + g(n)$ is $O(h(n) + p(n))$.
- **Product rule**: If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(p(n))$, then $f(n)g(n)$ is $O(h(n)p(n)$.
- **Polynomial rule**: If $f(n)$ is a polynomial of degree k, then $f(n)$ is $O(n^k)$.
- **Log of a power rule**: $log(nk)$ is $O(log(n))$ for any constant $k > 0$.

Coefficient Rule:

If $f(n)$ is $O(g(n))$, then $k(f(n))$ is $O(g(n))$, for any constant $k > 0$.

```
function x(n){
   var count =0;
     for (var i=0;i<n;i++){
       count+=1;
     }
  return count;
}
```

This block of code has $f(n) = n$ this is because it adds to count n times. Therefore, this function is $O(n)$. Here's another example code block for $kf(n)$:

```
function y(n){
  var count =0;
    for (var i=0;i<2*n;i++){
      count+=1;
    }
   return count;
}
```

This block has $kf(n) = 2n$. After all, the first two examples both have a Big-O notation of $O(n)$ or $O(g(n))$ from the above coefficient rule.

Sum Rule:

The sum rule is intuitive to understand — time complexities can be added. Imagine a master algorithm that involves two other algorithms — the Big-O notation of that master algorithm is simply the sum of the other two Big-O notations.

Note: It is important to remember to apply the coefficient rule after applying this rule.

Look at the code below:

```
 function z(n){
   var count =0;
   for (var i=0;i<n;i++){
    count+=1;
   }
   for (var i=0;i<5*n;i++){
    count+=1;
   }
   return count;
}
```

According to Sum rule it should be $f(n) = 1n$ and $g(n) = 5n$.

This results to $6n$ because $f(n) + g(n) = 6n$ but abiding to Coefficient rule it is $O(n) = n$ as $f(n)$ is $O(g(n))$, then $k(f(n))$ is $O(g(n))$.

Product Rule:

The product rule simply states how Big-Os can be multiplied.

The following code block demonstrates a function with two nested for loops (remember that this is a quadratic time inside product rule):

```
function (n){
  var count =0;
  for (var i=0;i<n;i++){
    count+=1;
    for (var i=0;i<2*n;i++){
      count+=1;
    }
  }
  return count;
}
```

In this example, $f(n) = 2nxn$ because the second loop has $2nxn$ which runs $2n$ times. Therefore, this results in a total of $2n$ operations. Applying the coefficient rule, the result is that $O(n) = n^2$.

Polynomial Rule:

The polynomial rule states that polynomial time complexities have a Big-O notation of the same polynomial degree.

If $f(n)$ *is a polynomial of degree k, then* $f(n)$ *is* $O(n^k)$.

The following code block has only one for loop with quadratic time complexity (quadratic time because $n * n$ is equal to 2 loop):

```
   function a(n){
    var count =0;
    for (var i=0;i<n*n;i++){
     count+=1;
    }
   return count;
}
```

In this example, $f(n) = n^2$ because the first loop runs $n * n$ iterations which is equivalent n^2 in accord to polynomial rule $f(n)$ is a polynomial of degree k, then $f(n)$ is $O(n^k)$.

Master Theorem

The master theorem states the following:

- Given a recurrence relation of the form $T(n) = aT(n/b) + O(n^c)$
- Where $a >= 1$ and $b >= 1$

a is the coefficient that is multiplied by the recursive call. b is the logarithmic term, which is the term that divides the n during the recursive call. Finally, c is the polynomial term on the nonrecursive component of the equation.

The first case is when the polynomial term $O(n^c)$ is less than $log_b(a)$

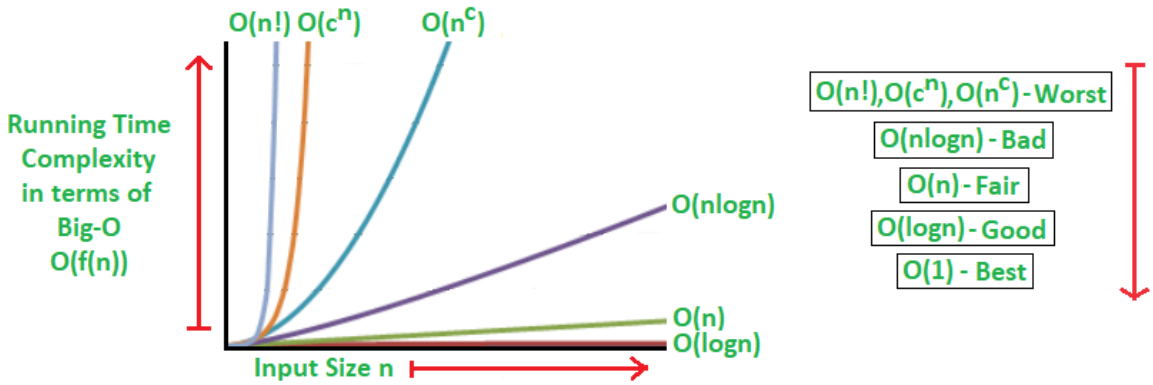
Case 1: If $c < log_b(a)$ then $T(n) = O(n^{(log_b(a))})$

Case 2: If $c = log_b(a)$ then $T(n) = O(n^c log(n))$

Case 3: If $c > log_b(a)$ then $T(n) = O(f(n))$

Things you learnt:

- Analysis of an algorithm with Big-O notation.
- Adding up Big-O Notations.
- Multiplying Big-O Notations.
- Master Theorem.



[1]

[1] Credit: www.geeksforgeeks.org