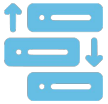
Algorithms: Introduction to Algorithmic Analysis (Part 1)

Topic to be covered

Introduction

Order of Growth

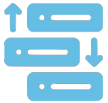
Asymptotic Analysis



Asymptotic Analysis

Time Complexity

Big O notation



MCQs

Topic to be covered

- Introduction
- Order of Growth
- Asymptotic Analysis
- Various types of Complexity Notations
- Big O notation
- MCQs
- Practice Problems

Introduction (10 mins)

Why do we need to analyze algorithms? Let's say you have two different algorithms performing the same task. How will you decide which one is better? What will be the parameters will you use to compare them, and how will you measure their performance?

Basically, we will always run any algorithm on a machine that incurs a cost (both operational as well as purchase), which we would like to minimize. Hence we would always want an algorithm that minimizes operational cost by running faster to completion (in fewer CPU cycles as possible) and consumes fewer resources like main memory. Hence it is pretty evident that time and space are the main two parameters on which we would compare any two algorithms.

To measure the performance of an algorithm one way to do it is via running it on a machine and measuring the time in seconds and space in terms of bytes consumed. This is popularly known as the experimental analysis of algorithms. But it has the following limitations:

- The parameters you are going to measure will have different values depending upon the type of machine it is running on, measurements are hardware dependent.
- Tricks played by the OS in the background like context switching, etc, are very hard to account for.
- You actually have to implement all algorithms before concluding which is better.

Due to these limitations, we generally rely on theoretical analysis of algorithms. Asymptotic analysis is the most popular theoretical analysis for measuring algorithmic complexity.

Order of Growth (15 mins)

Let's consider a single variate polynomial function f(N) = N^2 + N + 1. We know that the growth of the function is determined by the highest order term. In this case, the highest order term is N^2, therefore the growth rate of this function is N^2. We ignore lower order terms like N, 1 because, for larger values of N, N^2 will be much bigger than N, and will grow quadratically.

Order of growth deals with the growth rate of the function. **Order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example N, 10N, N+100, all have the same growth rate which is linear, as all of them grow linearly. Hence all of them fall under the same order of growth i.e. linear.

We use the same concept of order of growth in algorithmic analysis to basically rank them on basis of their growth rate measured in terms of input to the algorithm. The reason we measure it in terms of input is because for any algorithm the only entity that is bound to change is the input, making it the only variable in the system.

Order of Growth (15 mins)

Let's say we need to write a function to print first N natural numbers.

```
function printNaturalNumbers(N){
  for(let i = 1;i <= N; i++){
    console.log(i);
  }
}
```

Here N is input to the function. If we consider **console.log(i)** statement takes one unit time, then the function **printNaturalNumbers** will take N * 1 = N unit time. Therefore the time taken by the function depends **linearly** on input N.

Order of Growth (continued)

Let's say we need to write a function to print a 2D matrix with number of row = N, and number of cols = N;

```
function printMatrix(matrix, N){
    for(let i = 0;i < N; i++){
        for(let j = 0;j < N; j++){
            console.log(matrix[i][j]);
        }
    }
}
```

Here we have a nested for loop iterating over each cell of the 2D matrix.
Time taken for this function to execute
= time taken for **console.log() * number of cells**
= 1 * (number of rows * number of columns) //considering **console.log()** takes only one unit time
= 1 * N * N
= N^2
Therefore the order of growth of time for this function is **quadratic**.

Order of Growth (continued)

Some of the most commonly encountered order of growth functions are:
N^N, N!, 2^N, N^2, N√N, NlogN, NloglogN, N, √N, logN, constant.

For large values of N following is the relationship among the growth functions:
N^N > N! > 2^N > N^2 > N√N > NlogN > NloglogN > N > √N > logN > constant

Asymptotic Analysis (15 mins)

Asymptotic analysis is a study to **estimate the time and space requirements of a program as a function of input size**. Here we usually ignore cases when the input size is small since we are more interested in how the program will behave with larger inputs (N > 100).

In almost all the cases it is considered that the **lower the asymptotic growth rate better is the algorithm**. For example, consider f(N) as the asymptotic growth rate of the program, N is input size, c is a constant. Then **f(N) = c * N** denotes the linear growth rate of algorithmic complexity whereas for **f(N) = c * N^2**, the growth rate is **quadratic**.

Algorithmic complexity is a very important measurement that helps us to answer questions like

- How long will the algorithm run before producing output?
- How much space it will take?
- Is it possible to run the program with the given time and space constraints etc?

For example, if we are asked to write a program to store all the valid states of a standard 19 X 19 Go board. It is estimated the number of valid states is approximately 10^170 whereas the number of atoms in the observable universe is approximately 10^80, which means it is infeasible to run the program. The same is true for time complexity. If we assume generating each valid state requires one nanosecond (10^-9), then the program will take approximately 10^161 secs while the age of the universe is approximately 10^16 secs.

Types of Complexity Notations (30 mins)

Algorithmic complexity or the asymptotic growth rate has broadly three different types of notations:

1. **Big O notation** represented by "**O**" for measuring complexity in a worst-case scenario
2. **Theta notation** represented by "**θ**" for measuring the average complexity of the algorithm
3. **Omega notation** represented by "**Ω**" for measuring complexity in best case scenario

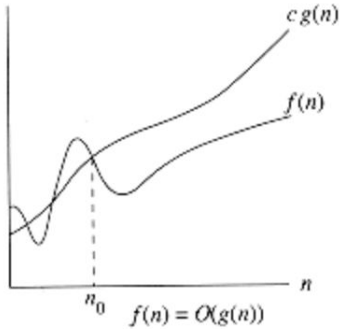
These notations are valid for both time and space complexity analysis. The difference between these measurements is important because a program can behave differently based on the nature of the input. One of the popular examples of this is Bubble sort where the best case time complexity is Ω(N) and the worst case is O(N^2) where N is the size of the list to be sorted.

Types of Complexity Notations (continued)

Big-O Notation (O)
It describes the **upper bound of algorithmic complexity**. For example, let the complexity of a program be represented via the following recurrence function: **f(N) = a*N + b*N^2 + c**, where N is the size of the input and a = 0 if N is even and b = 0 if N is odd and c is a constant. Then the Big-O complexity of the program is **O(f(N)) = O(a*N + b*N^2 + c) = O(k*N^2) = O(N^2)** where k is a finite positive number.

The formal definition of Big-O Notation is as follows:
f(n) = O(g(n)) if there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0

Graphical representation of Big-O Notation:

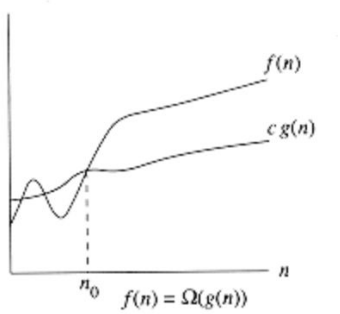


Types of Complexity Notations (continued)

Omega Notation (Ω)
It describes the **lower bound of algorithmic complexity**. For example, let the complexity of a program be represented via the following recurrence function: **f(N) = a*N + b*N^2 + c**, where N is the size of the input and a = 0 if N is even and b = 0 if N is odd and c is a constant. Then the Omega complexity of the program is **Ω(f(N)) = Ω(a*N + b*N^2 + c) = Ω(k*N) = Ω(N)** where k is a finite positive number.
The formal definition of Omega Notation is as follows:

f(n) = Ω(g(n)) if there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0

Graphical representation of Omega Notation:

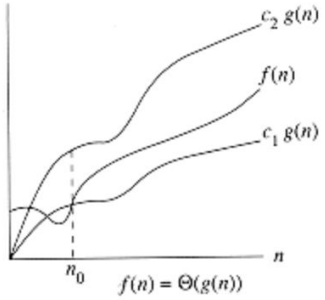


Types of Complexity Notations (continued)

Theta Notation (θ)
It describes both the upper and the lower bound of algorithmic complexity. For example, let the complexity of a program be represented via the following recurrence function: **f(N) = a*N + b*N^2 + c**, where N is the size of the input and a = 0 if N is even and b = 0 if N is odd and c is a constant. Then the theta complexity of the program is **k1*N <= θ(f(N)) <= k2*N^2** where k1 and k2 are finite positive numbers.
The formal definition of Theta Notation is as follows:

f(n) = Θ(g(n)) if there exist positive constants c1, c2 and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0

Graphical representation of Theta Notation:



Types of Complexity Notations (continued)

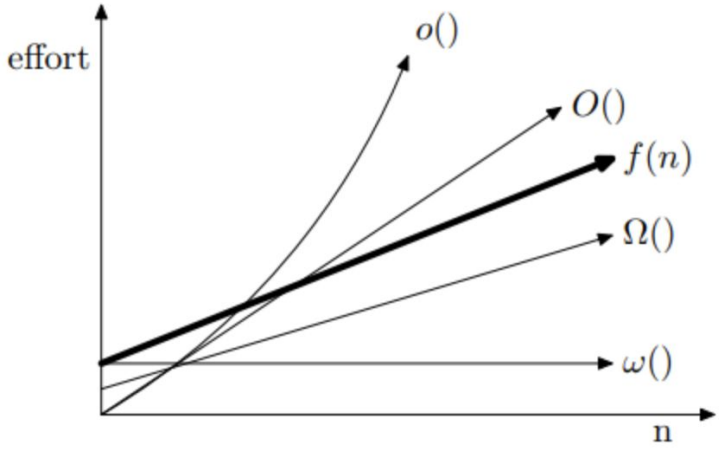
Apart from the above mentioned notations, there are two more which are slight variations of Big-O and Omega Notation

- **Little O**, represented by "**o**": It denotes loose upper bound, which is different from Big-O which denotes tight upper bound. For example, let f(n) = n + 2. From our knowledge of Big O notation, we can say that O(f(n)) = O(n). But in terms of Little o notation o(f(n)) = o(n^2). Little o always excludes the case of exact bound.
  Formal definition of little o notation is as follows:
  f(n) = o(g(n)) if there exist positive constants c and n0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n0$

- **Little Omega**, represented by "**ω**": It denotes asymptotically loose lower bound. For example, consider f(n) = n + 2. Therefore we know that Ω(f(n)) = Ω(n). But in terms of little omega notation, ω (f(n)) = ω(1). Just like little o, little omega also excludes the case of tight upper bound.
  Formal definition of little omega is as follows:
  f(n) = ω(g(n)) if there exist positive constants c and n0 such that $0 \leq cg(n) < f(n)$ for all $n \geq n0$

Types of Complexity Notations (continued)

Graphical representation of the relation between Little Omega, Little O with Omega, and Big O Notation:



Big O Notation (30 mins)

Consider a situation where you came up with an algorithm to solve a unique problem. You are confident about its correctness, but not sure about its efficiency. The most basic question that you will ask about your algorithm is how will it behave in the worst-case scenario of input. Basically how much your algorithm can scale with the increasing size of the input. Big O notation tries to answer this question on a very high level, without even the need to implement the algorithm in the first place.

Big O notation is all about finding the **asymptotic upper bound** of any algorithm's complexity. Big O notation is used to give an upper bound on a function, within a constant factor. Big O notation gives a high-level overview of how fast your algorithm is going to run as the input increases.

The **formal definition of Big O Notation** is as follows:
f(N) = O(g(N)) if there exists positive constants c, N0 such that f(N) <= c*g(N) for all N > N0. Here f(N) is upper bounded by g(N) such that f(N) grows no faster than g(N).

For example if **f(N) = N + N^2 + 2** then we can say for **N > 0, k = 4 and g(N) = k*N^2, f(N) <= g(N)**. Therefore **O(f(N)) = O(k*N^2) = O(N^2)**. Note that we ignore constants which do not depends upon input size N in our final calculation.

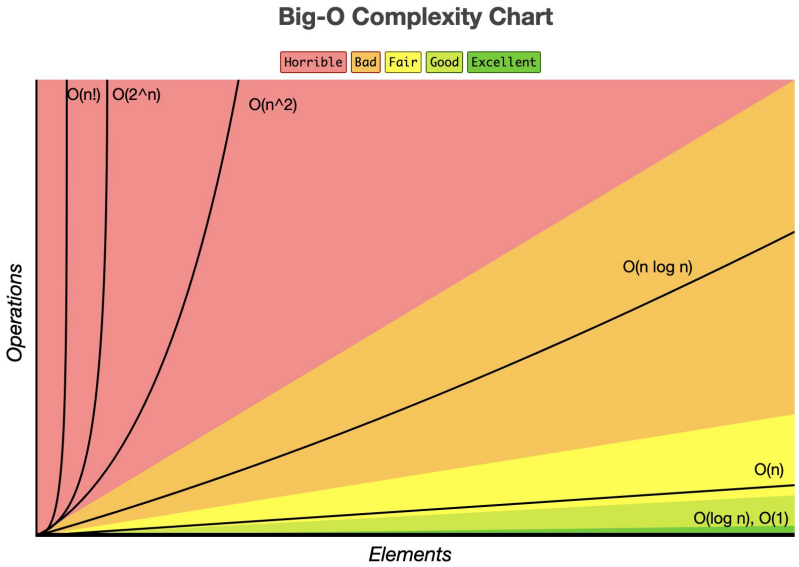
When we talk about algorithmic complexity we are only concerned about **tight bound**. For example, if f(N) is upper bounded by g(N) = N^2 then it is also upper bounded by g(N) = N^3. But we take the tight upper bound and report Big-O of f(N) as O(N^2).

Big O Notation (continued)

Common order of growth (in increasing order) seen in complexity analysis in term of Big-O notation is:

- O(1) : constant
- O(log N): logarithmic
- O(N) linear
- O(NlogN)
- O(N^2) quadratic
- O(N^3) cubic
- O(N^k) polynomial (for some positive integer k)
- O(C^N) exponential (for some positive number C)
- O(N!) factorial
- O(N^N)

Big O Notation (continued)



Big O Notation (continued)

Important properties of the Big-O notation

- Constant are always ignored. For all K > 0 O(Kf(n)) = O(f(n))
- Polynomial growth rate is determined by the leading term in the polynomial. Example f(n) = n^3 + n^2 + n + 1, then O(f(n)) = O(n^3)
- Transitive property holds with Big O notation. If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n)) is true.
- Product of upper bound is upper bond of the product. If f(n) = O(g(n)) and h(n) = O(r(n)), then f(n)*h(n) = O(g(n)*r(n))

Some general rules to follow while calculating Big-O complexity

- Always consider large values of n while comparing between two time complexity.
- O(1) has the least complexity also called constant complexity.
- Exponentials have greater complexity than polynomial. For example O(n^50) < O(2^n).
- Factorials have greater time complexity than exponentials.

Big O Notation (continued)

Short comings of Big-O notation:

- Many times the algorithm with the best Big O complexity is more difficult to read, understand and implement, and hence some times Big O complexity is ignored when not dealing with critical systems where every ounce of performance is necessary.
- Consider a hypothetical algorithm where running time is of order N in all the cases but just in one case, its complexity shoots to N^2. Big O notation will report it as O(N^2). Now let's say you come up with another algorithm for the same task which has a running time of order N√N. Big-O notation will report time complexity as O(N√N). Hence even though the first algorithm outperforms the second in almost all the cases still the second algorithm will be preferred. Hence many times people prefer average time complexity rather than worst-case complexity.

MCQ

1. Consider the following growth rate function in terms of input size (n): 10 ,√n ,n, log2n, 100n. Which of the following is the correct arrangement of the given growth functions in non decreasing order of growth rate.

A. log2n, 100n, 10,√n, n

B: 100n, 10, log2n, √n, n

C: 10, 100n, √n, log2n, n

D: 10, log2n, √n, n, 100n

Answer: D

2. Consider the following functions:

f(n) = n^2

g(n) = n!

h(n) = nlogn

Which of the following statements about the asymptotic behaviour of f(n), g(n), and h(n) is true?

A. f (n) = O(g(n)); g(n) = O(h(n))

B: f (n) = Ω(g(n)); g(n) = O(h(n))

C: g(n) = O(f (n)); h(n) = O(f (n))

D: h(n) = O(f (n)); g(n) = Ω(f (n))

Answer: D

MCQ

3. _____ is the formal way to express the upper bound of an algorithm's running time.

A. Omega Notation

B. Theta Notation

C. Big Oh Notation

D. All of the above

Answer: C

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

4. Which of the following is linear asymptotic notations?

A. O(1)

B. O(log n)

C. O(n)

D. O(n log n)

Answer: C

Explanation: Linear O(n)

MCQ

5. Omega notation is the formal way to express the lower bound of an algorithm's running time:

A. TRUE

B. FALSE

C. Can be true or false

D. Can not say

Answer: A

Explanation: True, Omega Notation is the formal way to express the lower bound of an algorithm's running time.

Practice Problems

1. Calculate the time complexity of Euclid's algorithm by subtraction
2. Calculate the factorial of a given number recursively and find its time and space analysis

Teaser for Upcoming Class

- Time and Space complexity calculation in detail

Thank You!