What is meant by Algorithm Analysis?

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why is Analysis of Algorithms Important?

1. To predict the behaviour of an algorithm without implementing it on a specific computer.
2. It is much more convenient to have simple measures for the efficiency of an algorithm.
3. It is impossible to predict the exact behaviour of an algorithm because there are too many influencing factors.
4. The analysis is thus only an approximation; but not perfect.
5. Mainly, by analysing different algorithms, we can have a comparison to determine the best one for our purpose

Types of Algorithm Analysis:

1. Best Case
2. Worst Case
3. Average Case

Introduction to Asymptotic Notation

Definition :

Asymptotic Notations are the mathematical tools to represent the Time complexity of algorithms for Asymptotic Analysis.

Asymptotic Analysis:

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by the programs to be compared.

1. No dependency on the machine.
2. We do not have to implement all algorithms.
3. It is about measuring order of growth in terms of input size.

There are mainly three asymptotic notations:

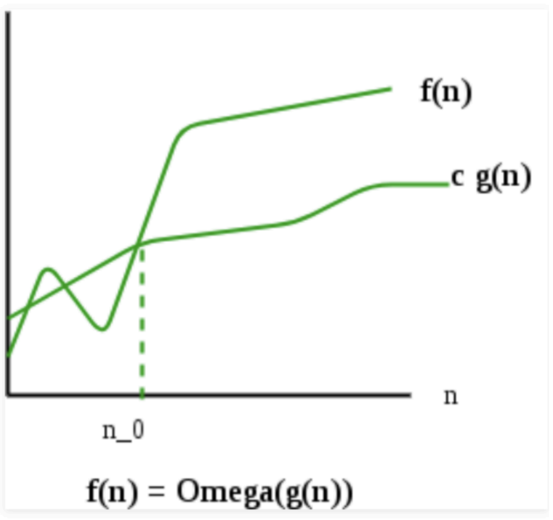
1. Big-O Notation
2. Omega Notation
3. Theta Notation

Big-O Notation (Upper Bound on Order of Growth):

The Big O Notation defines an upper bound of an algorithm, it bounds a function only from above.
It generally represented as f(n) = O(g(n))

We say f(n) = O(g(n)) iff there exist constants c and n_0 such that f(n) <= c*g(n) for all n >= n_0

n_0 is called the threshold for the given function.



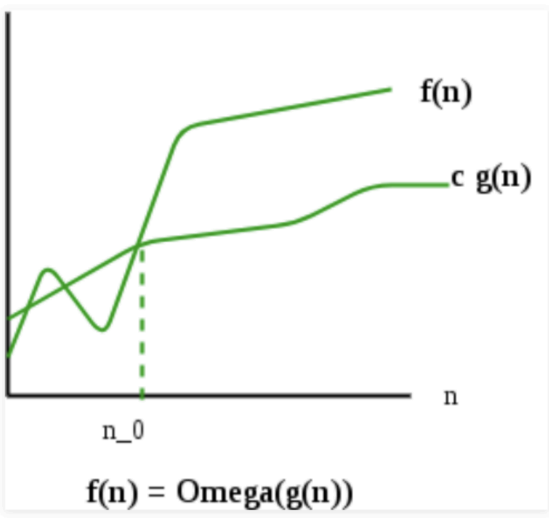
Note: g(n) is an asymptotic tight upper bound for f(n)
So our main objective is to give the smallest rate of growth g(n) which is greater than or equal to the given algorithm's rate of growth f(n).

we discard the lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm.

Omega Notation (Lower Bound on Order of Growth):

The Omega Notation defines a lower bound of an algorithm. It is useful when we have a lower bound on time complexity.

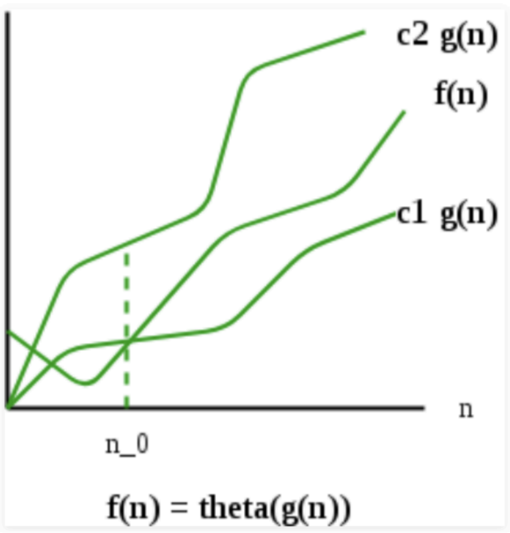
We say Ω(g(n)) = {f(n): there exists positive constants c and n_0 such that 0<= c*g(n) <= f(n) for all n >= n_0.
Our objective is to give the largest rate of growth g(n) which is less than or equal to the given algorithm’s rate of growth f(n).



Theta Notation (Exact Bound on Order of Growth):

This notation decides whether the upper and lower bounds of a given function are the same.

Θ(g(n)) = {f(n): there exist positive constants c_1, c_2 and n_0 such that 0 <= c_1*g(n) for all n >= n_0} is an asymptotic tight bound for f(n)*Θ(g(n)) is the set of functions with the same order of growth as g(n).



Best, Worst, Average Case Time Complexities

Worst Case(Usually Done): we calculate the upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. Linear search for a number that doesn't exist.

Average Case (Sometimes Done): we take all possible inputs and calculate computing time for all of the inputs.
Sum all the calculated values and divide by the sum of total number of inputs.

Best Case (Bogus): we calculate lower bound on running time of an algorithm. We must know the case that causes the minimum number of operations to be executed. In linear search for a number that appears in first place.

Analysis of Common Loop

While loop is a powerful construct in python that allows us to repeat a block of code as long as a specific condition remains true.

```
def add(a, b):
    result = a
    while b > 0:
        result += 1
        print(result)
        b -= 1

a = 10
b = 7
add(a,b)
```

Time complexity : O(b)

```
# code
def subtract(a, b):
    result = a
    while result >= b:
      print(result)
      result -= b

a = 15
b = 7
subtract(a, b)
```

Time complexity : O(a/b)

```
def multiply(n, c):
    result = 1
    while result < n:
      print(result)
      result *= c

n =32
c = 2
multiply(n, c)
```

Time complexity : O(log n)

```
def exponentfun(n, c):
    result = 2
    while result < n:
      print(result)
      result =result** c

base = 2
exponent = 32
exponentfun(exponent,base)
```

Time complexity : O(log log n)

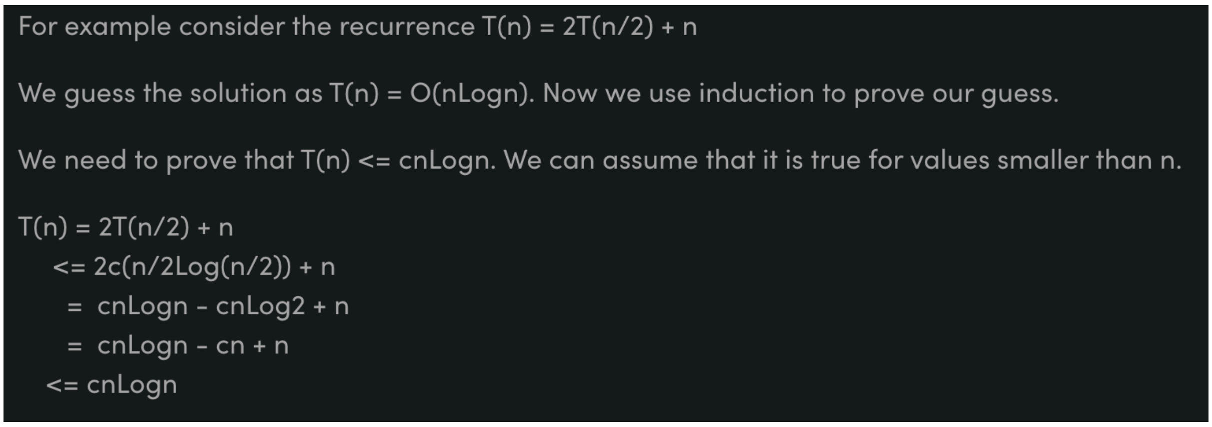
Analysis of Common Loop

Many algorithms are recursive. When we analyse them, we get a recurrence relation for the time complexity. We get running time on an input size n as a function of n and running time on inputs of smaller sizes.

Two methods

1. Substitution
2. Recurrence Tree

Substitution Method:



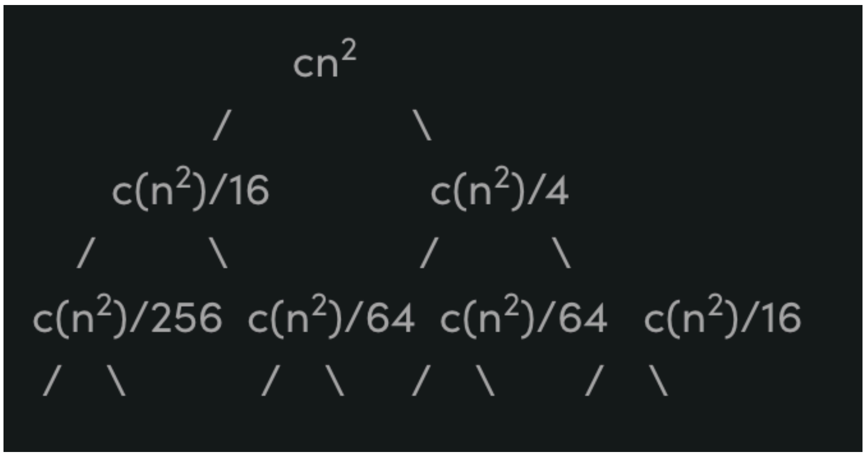
For example consider the recurrence $T(n) = 2T(n/2) + n$

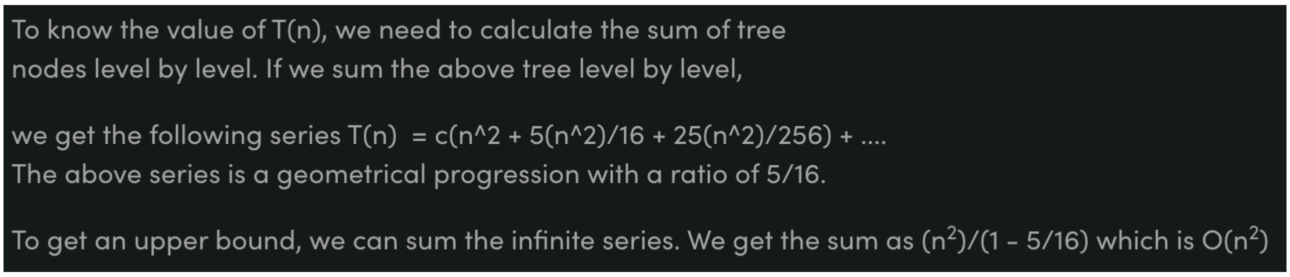
We guess the solution as $T(n) = O(nLogn)$. Now we use induction to prove our guess.

We need to prove that $T(n) <= cnLogn$. We can assume that it is true for values smaller than n.

```
T(n) = 2T(n/2) + n
    <= 2c(n/2Log(n/2)) + n
      =  cnLogn - cnLog2 + n
      =  cnLogn - cn + n
    <= cnLogn
```

Recurrence Tree Method:



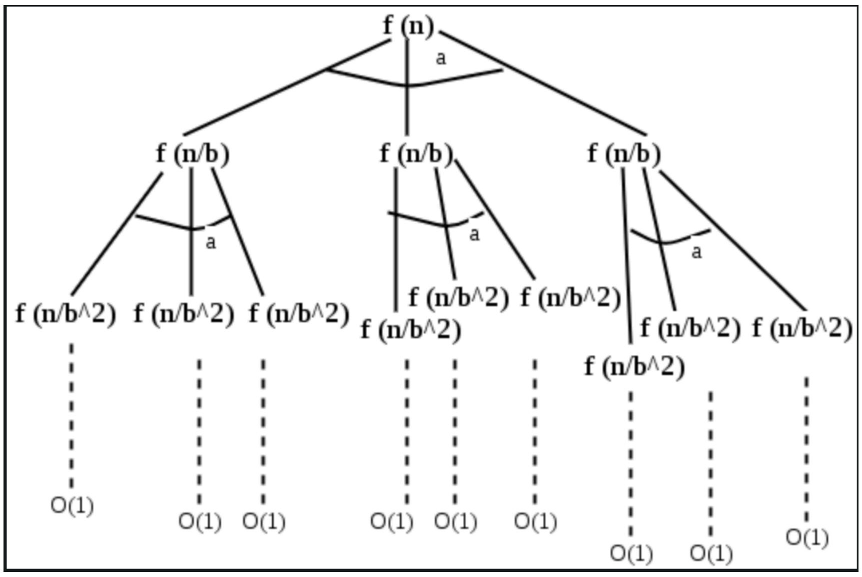


To know the value of T(n), we need to calculate the sum of tree
nodes level by level. If we sum the above tree level by level,

we get the following series $T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...$
The above series is a geometrical progression with a ratio of 5/16.

To get an upper bound, we can sum the infinite series. We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

Master Method :
It is a direct way to get the solution. It works only for the following type of recurrences or for recurrences that can be transferred into following type:
$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$



Space Complexity

The term Space Complexity is misused for Auxiliary Space at many places.

Auxiliary Space is the extra space used by a function.
Space complexity is the total space taken by the algorithm

```
def add(n):
  if(n<=0):
    return 0
  return n + add(n-1)
```

Aux Space : O(n)