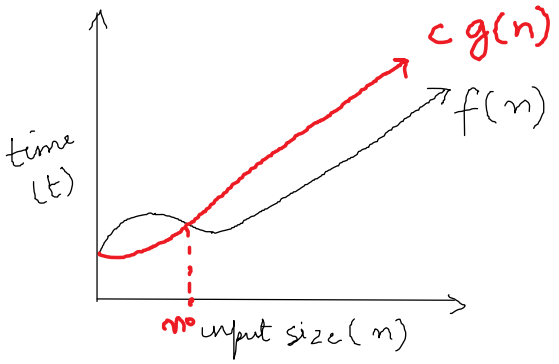
Algorithms

INTRODUCTION TO ASYMPTOTIC NOTATION

There will be many solutions to a given problem each of which will be given by an algorithm. Now, we want that algorithm which is going to give me the result quickest while consuming not much of the space (memory). Therefore, design and analysis of algorithms is the subject to design the various algorithms for a problem and analyze them for the best possible algorithm to choose.

Before proceeding, we have some notations to understand. These are called asymptotic notations.

Let's say we have a function f(n) and as the n increases the rate of growth of time increases in a certain way.



1. **Big-Oh (O)**: Let's find another function 'cg(n)' (above graph) in such a way that the after it gets an input n_0 the value of this function is always greater than the f(n). Then if:

$$f(n) \leq cg(n)$$
$$n \geq n_0$$
$$c > 0, n \geq 1$$

If the above satisfies, then we can say that

$$f(n) = O(g(n))$$

Which is equivalent to say that f(n) is smaller than g(n).

Let's take an example: f(n) = 3n + 2 and g(n) = n, is f(n) = O(g(n))?

$f(n) = 3n+2$, $g(n) = n$, for $f(n) = O(g(n))$ then

$\Rightarrow f(n) \leq c\,g(n)$

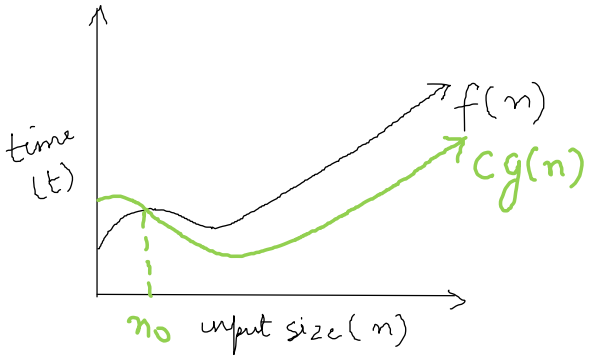
$\Rightarrow 3n+2 \leq cn$

Let $c=4 \Rightarrow 3n+2 \leq 4n \Rightarrow (2 \leq n) \therefore c=4$,

$\therefore f(n) = c\,g(n)$

Note: The tightest bound here is 'n'. If f(n) = O(g(n)) then definitely n^2, n^3… will be upper bounds, but we need to see the tightest bound which is 'n' in this case.

2. **Big-Omega(Ω)** :



Let's find another function 'cg(n)' (above graph) in such a way that the after it gets an input n_0 the value of this function is always smaller than the f(n). Then if:

$$f(n) \geq cg(n)$$
$$n \geq n_0$$
$$c > 0, n \geq 1$$

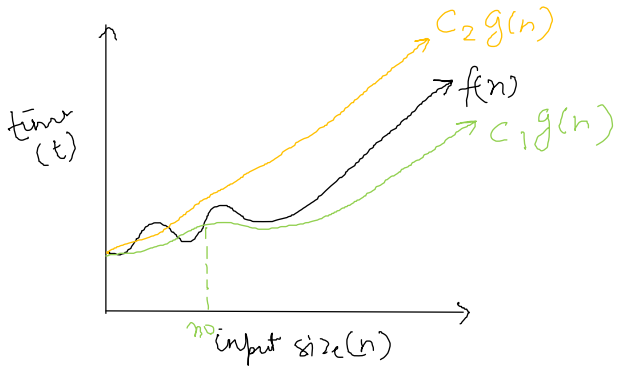
If the above satisfies, then we can say that

$$f(n) = \Omega\big(g(n)\big)$$

Which is equivalent to say that f(n) is greater than g(n).

We will see examples about it.

3. **Big-Theta(Θ):** We say that a function f(n) = Θ(g(n)), if the function f(n) is bounded by both lower and upper bounds. Let's see the graph for same.



Therefore,

$$c_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$C_1, C_2 > 0$$

$$n \geq n_0, n_0 \geq 1$$

If the above is satisfied then we can say f(n) = Θ(g(n))

Example: f(n) = 3n + 2, g(n) = n. Is f(n) = Θ(g(n))?

$f(n) = 3n+2$, $g(n) = n$

(1) $f(n) \le c_1 g(n)$

$3n+2 \le c_1 n$ (for $c_1 = 4$ this is true)

$\therefore 3n+2 \le 4n$ for all $n_0 \ge 1$

$\therefore f(n) = O(g(n))$ — (i)

(2) $f(n) \ge c_2 g(n)$

$3n+2 \ge c_2 n$

$3n+2 \ge n$ $(c_2 = 1)$

$\therefore 3n+2 \ge n$ for all $n_0 \ge 1$

$\Rightarrow f(n) = \Omega(g(n))$ — (ii)

(i) & (ii) $\Rightarrow$ $f(n) = \Theta(g(n))$

Practical significance of the symbols:

Given any algorithm:

- O is going to say "worst case time" or the "upper bound". In other words, if you are giving time complexity in O notation, that will mean "In any case the time will not exceed this".
- Ω is going to say "best case time" which means in any case you can never achieve better than this.
- Θ is giving you the average case

In practicality we worry about the question "What is the worst case time the algorithm will take for any input?"

For some algorithms, the best and worst case will be same. In those scenarios we go for Θ.

We have seen that an algorithm's time complexity is given as a function f(n). This function is an *approximation* of the time taken by the algorithm and NOT the actual time it may take. Now let's see how to find f(n) which means *what is the approximate time taken by the algorithm.* Before we go into that let's understand more about algorithms.

There are two types of algorithms:

1. Iterative
2. Recursive

- It is worth noting that *any program that can be written using iteration can be written using recursion and vice versa.* Therefore, both are equivalent in power.
- For iterative program, you count the number of times the loop is going to get executed to find the time it's going to take. To find the time taken by a recursive program we use recursive equations. This means that they are both same in power but their analysis methods are different.
- ***In case if the algorithm does not contain either iteration or recursion it means that there is no dependency of the running time on the input size which means whatever is the size of the input the running time will be a constant value.***

Let's see some of the iterative programs and how to do complexity analysis of them.

COMPLEXITY ANALYSIS OF ITERATIVE PROGRAMS

Example 1: Do the complexity analysis of the following program.

1)

```
A()
{
  int i,j;
  for(i=1 to n)
    for(j=1 to n)
      pf("Rahul")
}
```

n times, n times ∴ $O(n^2)$

2)

```
1991. A()
{ i=1, s=1;
  while (s<=n)
  { i++;
    s=s+i;
    pf("Rahul");
  }
}
```

Le 'k' be the value that 's' takes before the loop stops.

With each loop:

i=1, 2, 3, 4, … k

s = 1, 3, 6, 10, … n

We see that for every increase in 'i' s is the sum of first 'i' natural numbers. Therefore at when the loop reaches k to halt:

$K(k+1)/2 > n$

$(K^2 + k)/2 > n$

⇨ $K = O(\sqrt{n})$

3)

```
A()
{ for(i=1; i*i<=n; i++) ≡ for(i=1; i<=√n; i++)
  { pf("Hello");
  }
}
```

∴ $O(\sqrt{n})$

4)

```
A()
{
  int i, j, k, n;
  for(i=1; i<=n; i++)
  {
    for(j=1; j<=i; j++)
    {
      for(k=1; k<=100; k++)
      {
        Pf("ravi");
      }
    }
  }
```

from the following table:

innermost loop will get executed:

100 + 200 + 300 .... nx100 or

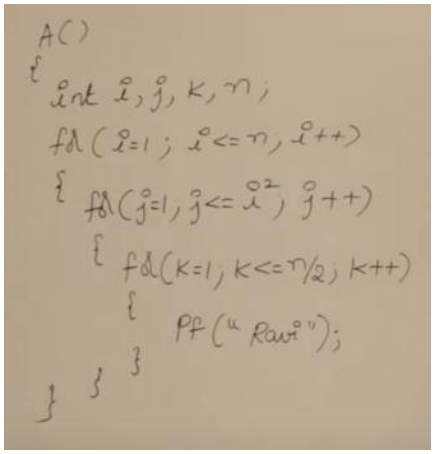
1x100 + 2x100 + 3x100 ... nx100 or

100 (1+2+3...n) or

100 (n(n+1)/2) ⇒ $O(n^2)$

I	1	2	3	4	…	N
J (times)	1	2	3	4	…	N
K(times)	1x100	2x100	3x100	4x100	…	Nx100

5



```
A()
{
  int i, j, k, n;
  fo(i=1; i<=n; i++)
  {
    fo(j=1; j<=i^2; j++)
    {
      fo(k=1; k<=n/2; k++)
      {
        Pf("Ravi");
      }
    }
  }
}
```

"Ravi" will get printed:

$n/2 \times 1 + n/2 \times 2 + n/2 \times 9 + n/2 \times 16 \cdots n/2 \times n^2$

$\Rightarrow n/2 \,(1 + 2 + 9 + 16 + n^2)$

$\Rightarrow n/2 \left(\frac{n(n+1)(2n+1)}{6}\right)$

$\Rightarrow \boxed{O(n^4)}$

I	1	2	3	4	…	N
J(times exec)	1	2	9	16	…	N^2
K(times exec)	(N /2)x1	(N /2)x2	(N /2)x9	(N /2)x16	…	(N/2)*N^2

$F(n) = n^k + n^{k-1} + n^{k-2} \ldots = O(n^k)$

6

```
A()
{ for (i=1 ; i<=n; i*=2)
  { print("Rahul"); }
}
```

Analysis:

for loops: $i = 1, 2, 4, \ldots, n$

Let us say it is going to take 'k' iterations by the time we reach 'n'.

∴ iterations: $2^0, 2^1, 2^2, 2^4 \ldots 2^k$

⇒ when program reaches 'n'

$2^k = n \Rightarrow \log(2^k) = \log(n)$

$\Rightarrow k = \log(n)$ ∴ $\boxed{O(\log n)}$

If i was increasing at the rate of 3, 4, 5, 6, instead of 2 in above example:

Then: for I = I * 2 : $O(\log_2 n)$

I = I * 3 : $O(\log_3 n)$ Therefore, for I = I *m complexity will be **$O(\log_m n)$**

I = I * 4 : $O(\log_4 n)$

I = I * 5 : $O(\log_5 n)$

Let's see an example depending on this example.

7

```
A()
{
  int i, j, k;
  fa(i=n/2; i<=n; i++)
    fa(j=1; j<=n/2; j++)
      fa(k=1; k<=n; k=k*2)
        Pf("ravi");
}
```

runs → $n/2$ times

runs → $n/2$ times

runs → $\log_2 n$ times (previous example)

$\therefore$ complexity $= n/2 * n/2 * \log_2 n$

$= O(n^2 \log_2 n)$

We don't need to unroll the loops in this example because every loop's conditional statement is independent of the variable of the other loops.

8

```
A()
{
  int i, j, k;
  fa(i=n/2; i<=n; i++)
    fa(j=1; j<=n; j=2*j)
      fa(k=1; k<=n; k=k*2)
        Pf(ravi);
}
```

runs → $n/2$ times

runs → $\log_2 n$ times

runs → $\log_2 n$ times

complexity: $n/2 * \log_2 n * \log_2 n$

$= O(n(\log_2 n)^2)$

In this example also the loops are independent of other loops' variables. So, no need to unroll them.

9

assume $n \geq 2$

```
A()
{ while (n>1)
  { n = n/2;
  }
}
```

Case 1: n is power of 2

for $n = 2^1$ while will execute 1 time

for $n = 2^2$ while will execute 2 time

for $n = 2^3$ while will execute 3 times

for $n = 2^k$ while will execute k times

$\therefore$ for n power of 2: $k = \log_2 n$

Case 2: n is not power of 2

let's say $n = 20$

→ 20 → 10 → 5 → 2 → 1 (4 times)

$\lfloor \log_2 20 \rfloor \simeq 4$

$\hookrightarrow O(\lfloor \log_2 n \rfloor)$

For question 9 above, if n was updating like:

N = n/5 then the complexity would be $O(\log_5 n)$

N = n/m then the complexity would be $O(\log_m n)$

Important result

10

```
A()
{
  for(i=1; i<=n; i++)
    for(j=1; j<=n; j=j+i)
      Pf("ravi");
}
```

from the following table:

"ravi" will get printed:

$N + N/2 + N/3 + \cdots + N/N$

$= N(1 + 1/2 + 1/3 + \cdots + 1/N)$

$= N(\log N) = O(n \log n)$

The inner loop is dependent on the variable of the outer loop. In this example we will have to unroll the loops to see wassup!

For I equals to	1	2	3	4	…	K	N
J will execute	1 to N	1 to N	1 to N	1 to N	…	1 to N	1 to N
J will execute times	N times	N/2 times	N/3 times	N/4 times	…	N/K times	N/N times

11

```
A()
{ int n = 2^k;
  for(i=1; i<=n; i++)
  {
    j=2
    while(j<=n)
    {
      j=j^2;
      Pf("ravi");
    }
  }
}
```

from the table below:

printf("ravi") will run

$n*(k+1)$ times

$n = 2^{2^k} \Rightarrow \log(n) = 2^k$

$\Rightarrow \log(\log(n)) = k$

$\Rightarrow n(\log\log n + 1) = n\log\log n + n = O(n \log\log n)$

At K	1	2	3	K
N will be	4	16	2^8	2^K
J values after each loop	2, 4	2, 4, 16	2, 4, 16, 256	2, 4, 16, …, 2^K
While will execute	N x 2 times	N x 3 times	N x 4 times	N x (K+1) times