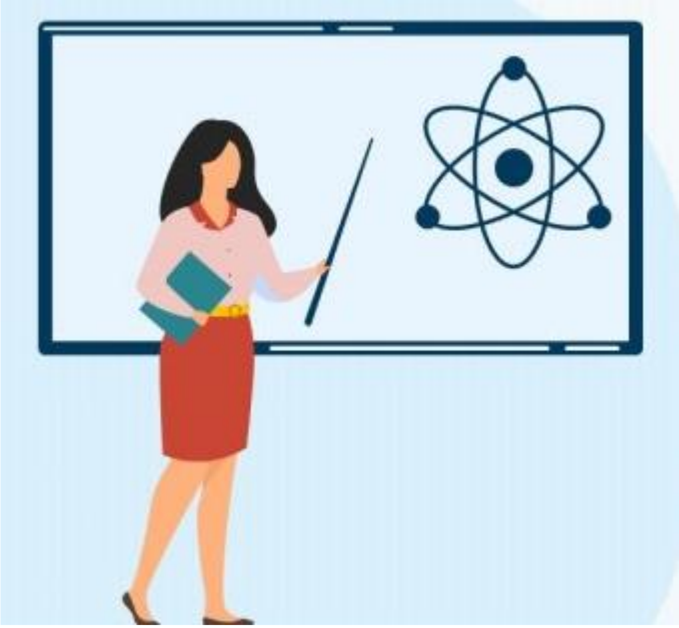
Algorithm Analysis



Suman Pandey

Agenda

- Experimental Studies
- Seven **Functions** Used in the Book
  - **This topic is highly mathematical**
  - **We will discuss various types of functions, that can represent a set of steps in algorithm**
- Asymptotic Analysis
  - **Theta**
  - **Big-Oh**
  - **Omega**
- Simple Justification Techniques with Examples

How would you analyze your Algorithm Performance?

- Experimental Analysis
- Asymptotic Analysis

Experimental Analysis

- We can analyze the running time of the algorithm simply with the difference of starting and ending time

```
from time import time
start_time = time( )                 # record the starting time
run algorithm
end_time = time( )                   # record the ending time
elapsed = end_time − start_time      # compute the elapsed time
```

- This has several drawbacks
  - In python you cant see fractions of millisecond times. It will show 0.0
  - This will vary from CPU to CPU and computer to computer
  - A part of the algorithm cant be verified

Asymptotic Analysis

- How to do analysis without performing Experiments ?

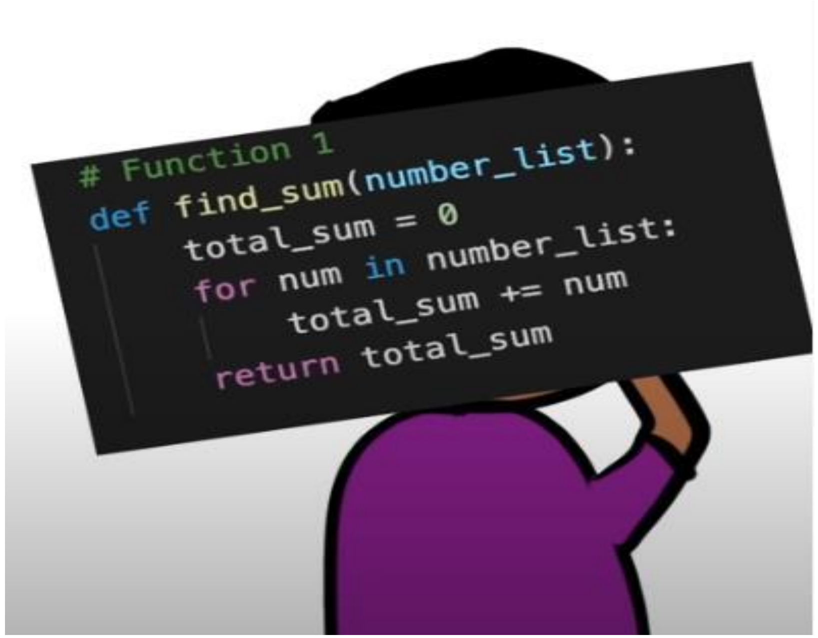
- We perform an analysis directly on a high-level description of the algorithm
  - In the form of actual code fragment Or language-independent pseudo-code
  - We define as a set of **Primitive Operations**
    - Assigning
    - Arithmetic Operation
    - Comparing
    - Accessing element in list
    - Calling a function
    - Returning a function

Count these basic operations executed by the hardware

t time

  - This count of primitive operations will correlate to an actual running time of algorithm
  - These operations will be higher if input is high
- No of operations will be higher if input is high
  - Hence it should be represented as function of Input size **f(n)**, input size is **n**

- Big O Notation
  - A mathematical notation used to classify algorithms according to how their **run time** or space requirements grow **as the input size grows**.



1	2	2	3	4	5	6	7

1	2	2	3	4	5	6	7

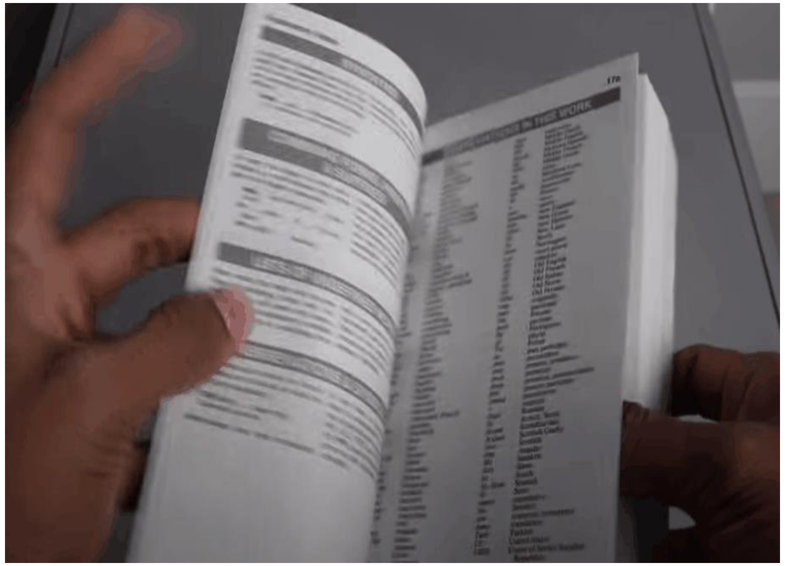
- 1 Operation - **O(1)**

n Operation - **O(n)**

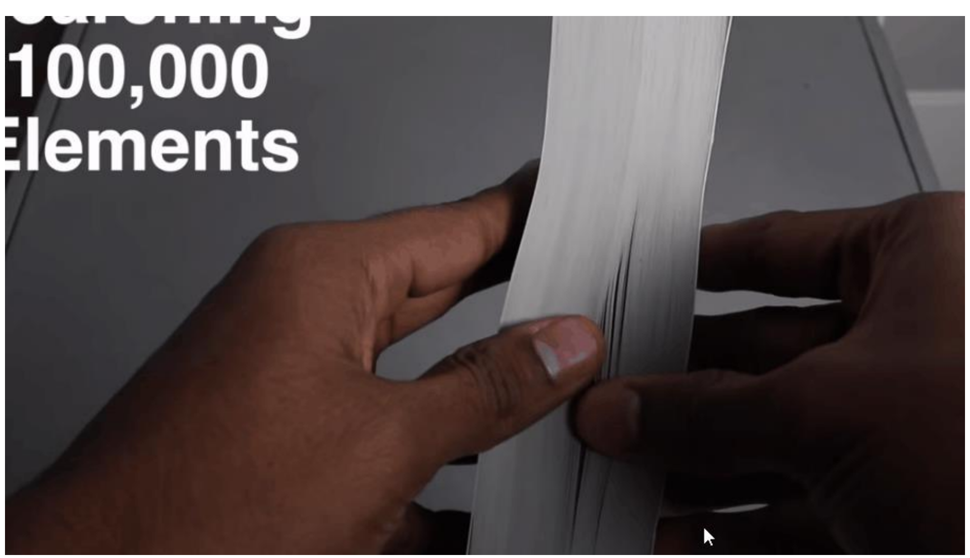
Big 0 Searching Dictionary

- Big 0 Notation

Sequential Search



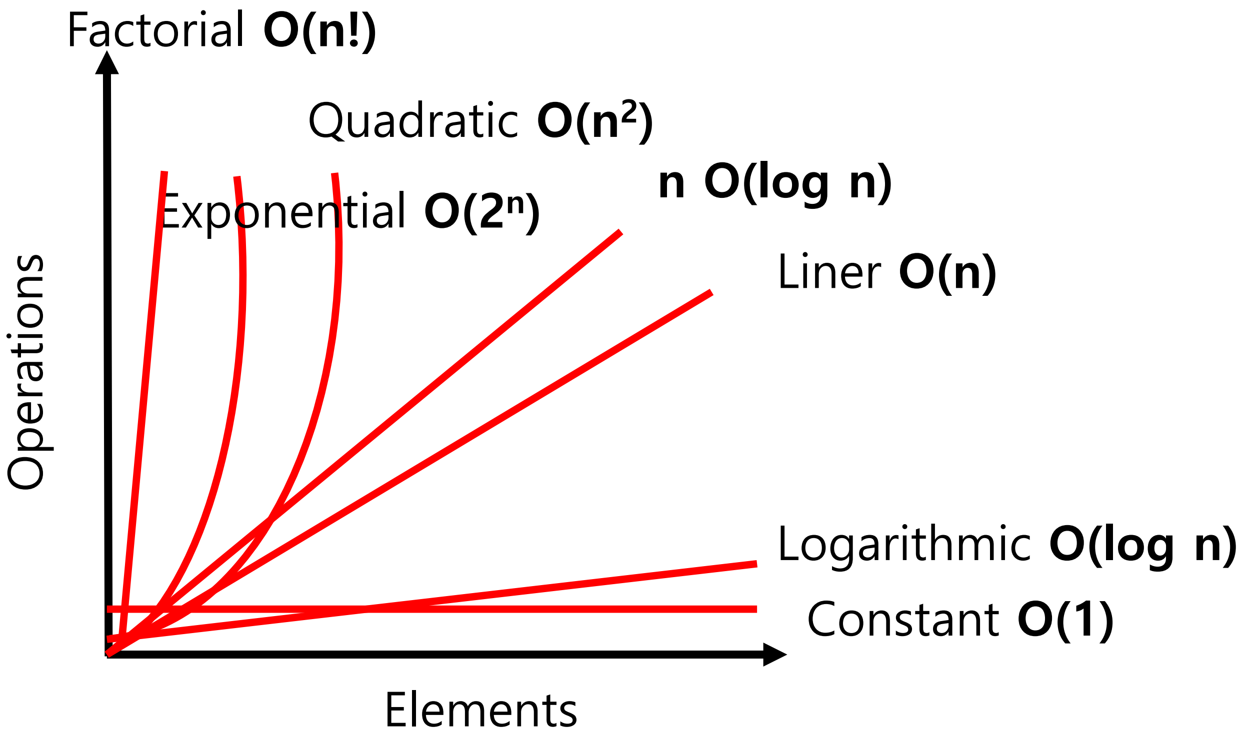
Binary Search



- 1 Operation - **O(n)**

n Operation - **O(log n)**

Time Complexities shown through 7 functions



N: 17

O(1):	1
O(Log N):	4
O(N):	17
$O(N^2)$:	289
$O(2^N)$:	131072
O(N!):	355687428096

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \mathinner{..} j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

$$\Downarrow$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

$$\Downarrow$$

$$an^2 + bn + c$$

In **Asymptotic notation**
We are only interested in how the running time of an algorithm increases with the size of the input in the limit

$\Rightarrow$

We ignore a, b c constants
We only consider an^2 , lower order are insignificant
We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared")

Example 1: What will be Time Complexity

```
for ( i =0; i< n; i++)
{
    stmt;    - n times
}
```

Ans : O (n)

```
for ( i = n ; i > 0; i--)
{
    stmt;    - n times
}
```

Ans : O (n)

```
for ( i = 1 ; i <n ; i=i+2)
{
    stmt;    - n/2
}
```

Ans : O (n/2) = O(n)

degree of the polynomial is 1, so it will be O(n)

Example 2: What will be Time Complexity of these

```
for ( i =0; i< n; i++)
{
    for ( j =0 ; j<n; j++)
    {
      stmt;
    }
}
```

- nxn times

Ans : O (n^2)

For value of i	Value of j	
0	n	n
1	n	n
2	n	n
.		
.		
n	n	n

Time Complexity → n*n = $\mathbf{n^2}$

Example 2: What will be Time Complexity of these

```
for ( i =0; i< n; i++)
{
    for ( j =0 ; j<i; j++)
    {
      stmt;
    }
}
```

Ans : O (n^2)

For value of i	Value of j	
0	0	0
1	0…1	1
2	0….2	2
3	0….3	3
.		
n	0….n	n

Time complexity → **n(n+1)/2**

1+2+3+… n = n(n+1) /2

F(n) = n^2 +1 /2

Order of polynomial is also n^2

Example 3: What will be Time Complexity of these

```
p=0
for ( i =1; p< n; i++)
{
    p=p+i ;
}
```

Ans : O(√ n)

Time Complexity →

When i	p is
1	0+1=1
2	1+2
3	1+2+3
4	1+2+3+4
.	
.	
.	
k	**1+2+3 … +k**

Assume loop exits at k

Loop will exit p >n

That is $p = k(k+1)/2$

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

$k^2 > n$

$k > \sqrt{n}$

$O(\sqrt{n})$

Example 4: What will be Time Complexity of these

```
for ( i =1; i< n; i= i*2)
{
    stmt
}
```

Blindly we cant say this loop will
Execute n times

Ans : O($\log_2$ n)

If you have a loop where counter variable is not incrementing but getting multiplied then the complexity will be log n

Q. Can you tell me what will be the time complexity of this

```
for ( i =1; i< n; i= i*3)
{
    stmt
}
```

Ans : O($\log_3$ n)

i

1
1x 2
2 x 2
4 x 2
..
..
2^k
Lets say this loop execute for k times
At k time i >n
That is i = 2^k
That is 2^k > = n
2^k = n
k = $\log_2$ n

Example 5: What will be Time Complexity of these

```
for ( i =  n ; i>=1 ; i= i/2)
{
    stmt;
}
```

Blindly we cant say this loop will Execute n times

Ans : $O(\log_2 n)$

If you have a loop where counter variable is not incrementing but getting dividing then the complexity will be log n

i is starting with n
---
n
$n/2$
$n/2^2$
$n/2^3$
..
..
$n/2^k$

We don't know number of time
We assume that the loop execute for k ti mes
At k time $i < 1$
That is $n/2^k < 1$
That is $n/2^k = 1$
$n = 2^k$
$k = \log_2 n$

Example 6: What will be Time Complexity of these

```
for ( i =0; i< n; i++)
{
    stmt;    - n times
}
for ( i =0; i< n; i++)
{
    stmt;    - n times
}
```

Ans : O (n)

```
p=0
for ( i =1; i< n; i=i*2)
{
    p++;     -> log n
}

for ( j =0; j < p; j=j*2)
{
    stmt;     -> log p
}
```

P = log n
log p = log log n
O(log n + log log n)
Ans: O(log n)

```
P=0
for ( i =0; i< n; i++)                 -> n
{
        for ( j =1; j< n; j=j*2)  -> n log n
        {
                stmt;  -> n log n
        }
}
```

Ans: n + 2 n log n -> **O (n log n)**

```
for ( i =0; i< n; i++)
{
    stmt;                 -> n
}

for ( j =0; j< k; j++)
{                          -> k
    stmt;
}
```

Ans: O(n + k)

Mathematical Explanation Of Asymptotic Notation

Asymptotic Notations are tools for comparing the growth rates of functions.

Asymptotic Analysis

- How to do analysis without performing Experiments ?

- We perform an analysis directly on a high-level description of the algorithm
  - In the form of actual code fragment Or language-independent pseudo-code
  - We define as a set of **Primitive Operations**
    - Assigning
    - Arithmetic Operation
    - Comparing
    - Accessing element in list
    - Calling a function
    - Returning a function

Count these basic operations executed by the hardware

t time

  - This count of primitive operations will correlate to an actual running time of algorithm
  - These operations will be higher if input is high
- No of operations will be higher if input is high
  - Hence it should be represented as function of Input size **f(n)**, input size is **n**

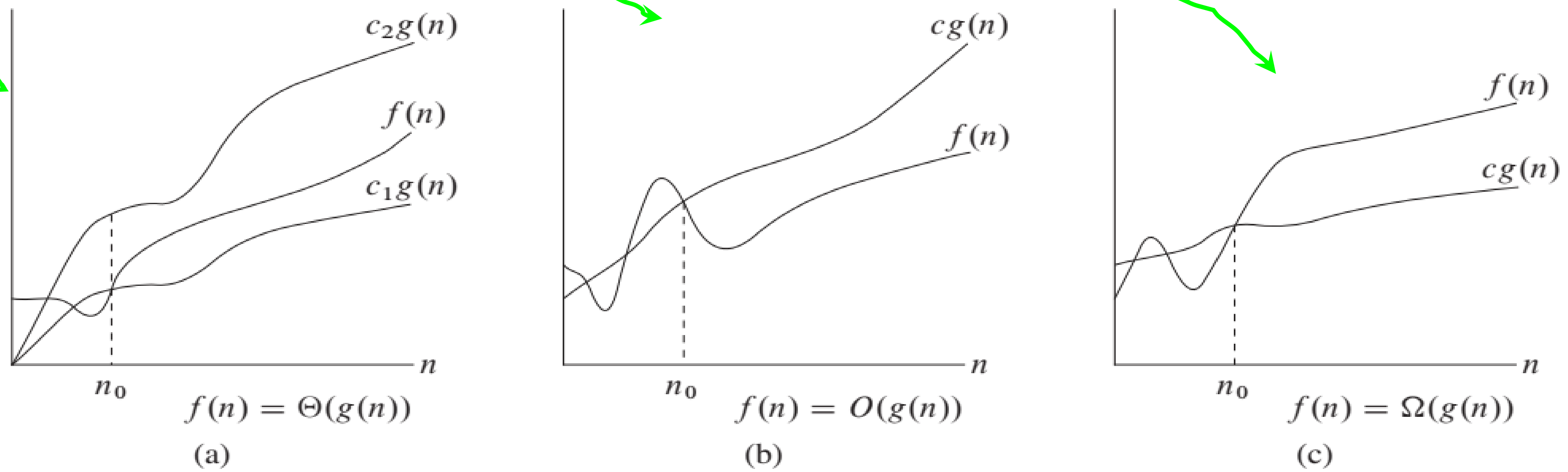
Notations

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ 3^n < \ldots\ldots\ n^n$

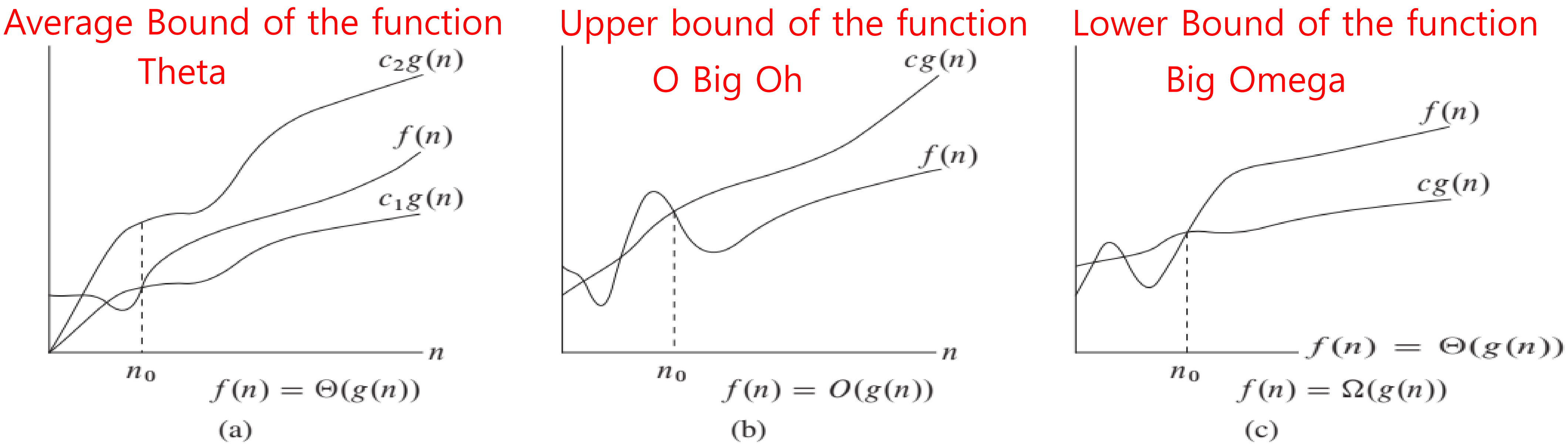
Time complexity is represented as one Among these or the multiple of these

- **Θ** Theta - Average Bound/tight bound of the function - **The best**
- **O** Big Oh - Upper bound of the function - When Theta is impossible use big Oh
- **Ω** Big Omega - Lower Bound of the function

Asymptotic means - approaching but never co nnecting with a line or curve



Mathematical Definition of Notations



We can also say that f(n) is member of Θ(g(n)), f(n) ∈ Θ(g(n))
we are abusing the equality in this way

Θ-notation — Theta notation bounds a function to within constant factors. The function f(n) = Θ(g(n)) if there are positive constants n_0, c_1 and c_2 such that $\mathbf{c_1g(n) \le f(n) \le c_2g(n)}$ for all $n \ge n_0$.

O-notation — Big Oh gives an upper bound for a function to within a constant factor. The function f(n) is O(g(n)) if there is a positive constant n_0 and c such that $\mathbf{f(n) \le cg(n)}$, for $n \ge n_0$.

Ω -notation — Omega gives a lower bound for a function to within a constant factor. The function f(n) = Ω (g(n)) if there are positive constants n_0 and c such that $\mathbf{f(n) \ge c\ g(n)}$ for all $n \ge n_0$

f(n)

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)\,.$$

$$\Downarrow$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

$$\Downarrow$$

f(n) $= an^2 + bn + c$

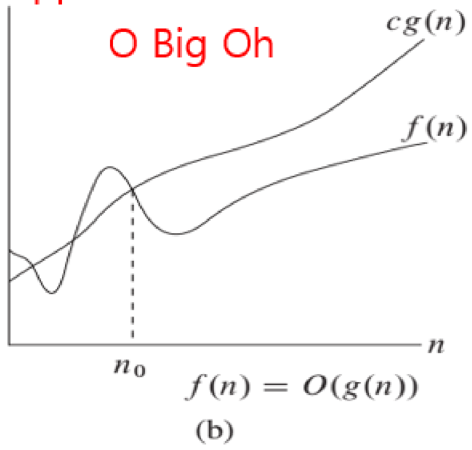
Time Complexity **O(n^2)**

But there is a Mathematical Explanation and Definition in next slide

O Big Oh

The function f (n) is O(g(n)) if there is a real constant c > 0 such that f(n) ≤ cg(n), for n ≥ n_0.

Upper bound of the function



Ex: **f(n) = 2n + 3**

if we want to write this statement *f*(*n*) ≤ *cg*(*n*)

2n + 3 ≤ ?? (what should go in ??)

?? Could be anything greater than 2n + 3

?? = 10n 10 is c, n is g(n) and 10n is greater than 2n+3 for all n ≥ 1

?? = 7n, 7 is c, n is g(n) and 7n is greater than 2n+3 for all n ≥ 1

What should we write ?

Better idea

?? = 2n + 3n = 5n, 5 is c, n is g(n) and 5n is greater than 2n+3 for all n ≥ 1 -> f(n) is O(g(n)) is **O(n)**

Other solutions

?? = n^2 (yes possible) O(n^2)

?? = n^3 (yes possible) O(n^2)

?? = n^n (yes possible) O(n^n)

But, we should try to keep cg(n) close to f(n)
Hence f(n) = O(n) is best solution

Lower Bound 1 < log n < √ n < n < n log n < n^2 < n^3 < ……. 3^n < …….. n^n Upper bound

Average Bound

Upper , Lower and Average Bound

Lower Bound $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ 3^n < \ldots\ldots\ n^n$ Upper bound

Average Bound / tight bound

Ω -notation

Ω -notation

The function f(n) = Ω(g(n)) if there are positive constants n0 and c such that f(n) ≥ c g(n) for all n ≥ n0

Ex: **f(n) = 2n + 3**

if we want to write this statement $f(n) \geq cg(n)$

2n + 3 **≥** ?? (what should go in ??)

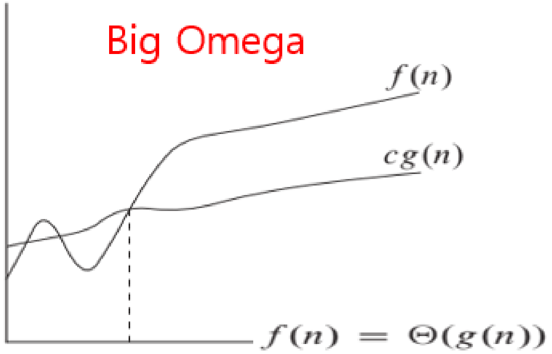
?? Could be anything less than 2n + 3

?? = 1xn , 1 is c, n is g(n) and 1n is smaller than 2n+3 for all n ≥ 1 -> f(n) is Ω(g(n)) is **Ω(n)**

Other solutions

2n + 3 **≥ 1 x log n** for all n ≥ 1 (yes possible) **Ω(log n)**

Lower Bound of the function



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

(c)

But, we should try to keep cg(n) close to f(n)
Hence f(n) = Ω(n) is best solution

Lower Bound $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ 3^n < \ldots\ldots\ n^n$ Upper bound

Average Bound

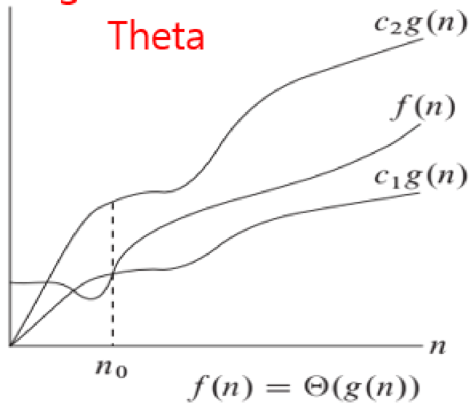
Θ -notation

Θ -notation

The function $f(n) = \Theta(g(n))$ if there are positive constants n_0, c_1 and c_2 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

Average Bound of the function

Theta



Ex: $\mathbf{f(n) = 2n + 3}$

if we want to write this statement $c_1 g(n) \le f(n) \le c_2 g(n)$

$1n \le 2n + 3 \le 5n$ (we already found the lower and upper bound in previous examples)

1 is c_1 5 is c_2 n is g(n)-> f(n) is $\Theta(g(n))$ is $\mathbf{\Theta(n)}$

Other solutions

we cant say f(n) is $\Theta(n^2)$

we cant say f(n) is $\Theta(\log n)$

Hence theta notation is tight bound , f(n) is only $\mathbf{\Theta(n)}$

This is best representation of time complexity

note: not all functions can be represented as theta notation.

Lower Bound $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots. \; 3^n < \ldots\ldots.. \; n^n$ Upper bound

Average Bound

Still don’t understand ? What is all this ???

- Lets understand with some examples

Example 1

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ 3^n < \ldots\ldots\ n^n$

$F(n) = 2n^2 + 3n + 4$

$2n^2 + 3n + 4 <= 2n^2 + 3n^2 + 4n^2$

$2n^2 + 3n + 4 <= \underset{c}{9}\ \underset{g(n)}{n^2}$ $\quad n >= 1$

$f(n) \le cg(n)$

$F(n) = \mathbf{O}(n^2)$

$F(n) = 2n^2 + 3n + 4$

$2n^2 + 3n + 4 >= 1n^2$

$f(n) \ge c\, g(n)$

$F(n) = \mathbf{\Omega}(n^2)$

$F(n) = 2n^2 + 3n + 4$

$1n^2 <= 2n^2 + 3n + 4 <= 9 n^2$

n^2 both the sides

$c_1 g(n) \le f(n) \le c_2 g(n)$

$F(n) = \mathbf{\Theta}(n^2)$

```
for ( i =0; i< n; i++)
{
   for ( j =0 ; j<n; j++)
   {
     stmt;
   }
}
```

n^2

```
for ( i =0; i< n; i++)
{
   for ( j =0 ; j<n; j++)
   {
     stmt;
   }
}
```

n^2

```
for ( i =0; i< n; i++)
{
              stmt;
}
```

n

```
for ( i =0; i< n; i++)
{
              stmt;
}
```

n

```
for ( i =0; i< n; i++)
{
              stmt;
}
```

n

```
stmt;
stmt;
stmt;
stmt;
```

4

Example 2

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ 3^n < \ldots\ldots\ n^n$

$c_1 g(n) \le f(n) \le c_2 g(n)$

$F(n) = n^2 \log n + n$

$1\ n^2 \log n <= n^2 \log n + n <= 10\ n^2 \log n$ 10 and 1 are any arbitrary value

$n^2 \log n$ both the sides

$F(n) = \boldsymbol{\Theta}(n^2 \log n)$ $F(n) = \mathbf{O}(n^2 \log n)$ $F(n) = \boldsymbol{\Omega}(n^2 \log n)$

where will this class fall in this spectrum ?

Example 3 : Where its not possible to find complexity in Θ

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots\ 3^n < \ldots\ldots\ n^n$

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

F(n) = n! = $n * (n-1) * (n-2) \ldots\ldots\ 3 * 2 * 1$

$2 * 2 * 2 * 2 <= 1 * 2 * 3 \ldots\ldots * n <= n * n * n * \ldots\ldots N$

$2^n <= n! <= n^n$

both the sides are not same

In nutshell, If you can not find any meaningful lower bound, in order of n then its not possible to define a Θ for that function

$F(n) = \boldsymbol{\Omega}(2^n)$ $F(n) = \mathbf{O}(n^n)$

F(n) = **Θ**(????) -> we cant find any value for theta notation, hence we need to go with big Oh or Omega

So we go for upper bound or lower bound

n! can not be expressed in the form of Θ

- In the place of 2 * 2 * 2 * 2….2 < = 1 *2 *3 …… * n < = n * n * n * ….. n
- You can think of any other mathematical hypothesis as well
- lower bound function g(n) that is proportional to n^k for some positive constant k (i.e., g(n) = O(n^k)).
- As **n** increases, the factorial function **n!** will grow much faster than any polynomial function of n (such as n^k). Think of n growing to significantly large value.
- n! grows so rapidly that it is difficult to find a lower bound that is both proportional to n! and informative. As a result, Θ notation is not typically used for factorial time complexities due to the absence of a well-defined lower bound.

o-notation / Ω -notation

- **abusing the notation of n**. We can think of n to grow ∞ . This can be expressed with the help **o-notation** (upper bound that is not asymptotically tight) and **Ω -notation** (lower bound that is not asymptotically tight)

o(g(n)) = {f(n): for any positive constant c> 0 and n_0>0 such that **0 <=f(n) < cg(n)** for all n >=n_0

in **o-notation**, the function f(n) becomes insignificant relative to g(n) as n approaches infinity; 2n = o(n^2), but 2n^2 != o(n^2)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Ω(g(n)) = {f(n): for any positive constant c> 0 and n_0>0 such that **0 <=cg(n)< f(n)** for all n >=n_0

in **Ω-notation**, the function f(n) becomes significant large relative to g(n) as n approaches infinity; $n^2/2$ = **Ω**(n), but $n^2/2$!= **Ω**(n^2)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

F(n) = 2n

Ω -notation 1 < log n < √ n <n < n log n < n^2 < n^3 < .... 3^n < .... n^n **o-notation**

Average Bound / tight bound

Properties of Asymptotic Notations

Transitivity:

$$
\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &&\text{imply} \quad f(n) = \Theta(h(n))\,, \\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) &&\text{imply} \quad f(n) = O(h(n))\,, \\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &&\text{imply} \quad f(n) = \Omega(h(n))\,, \\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) &&\text{imply} \quad f(n) = o(h(n))\,, \\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) &&\text{imply} \quad f(n) = \omega(h(n))\,.
\end{aligned}
$$

This means that if one function grows slower than anoth er, and that function grows slower than a third function, t hen the first function also grows slower than the third fu nction.

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))\,.$$

Meaning, when both the functions are same (this holds of theta)

ex: $f(n) = n^2$, $g(n) = n^2$

have the same growth rate, you can swap them in the notation.

Transpose symmetry:

$$
\begin{aligned}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))\,, \\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n))\,.
\end{aligned}
$$

This is true for big O and omega

Which means if f(n) is upper bound then g(n) is lower bound.

Comparing two functions

Asymptotic Notations are tools for comparing the growth rates of functions.
So sometimes you might need to compare two asymptotic notations.

$f(n) = n^2 \quad g(n) = n^3$

Ans: f(n) < g(n)

$f(n) \le cg(n)$ -> f(n) is O(g(n))

f(n) is O(g(n))

g(n) is Ω(f(n))

This was a simple function, but for complicated function you can take a log.

log (f(n)) = 2 log n

log (g(n)) = 3 log n

After applying log, both the function looks same, but after applying log we must not cancel coefficients.

log (f (n)) < log (g(n)) -> f (n) < g (n)

However we can say

log (f(n)) is Θ of log (g (n))

Comparing two functions

Asymptotic Notations are tools for comparing the growth rates of functions.
So sometimes you might need to compare two asymptotic notations.

$f(n) = n^2\log(n)$ $g(n) = n\,(\log n)^{10}$

$\log[n^2\log(n)]$ $\log[n\,(\log n)^{10}]$ Based on 1.

$\log n^2 + \log\log(n)$ $\log n + \log(\log n)^{10}$ Based on 2.

$2\log n + \log\log n$ $\log n + 10\log\log n$

Ans: f(n) > g(n)

1. $\log_c(ab) = \log_c a + \log_c b$,
2. $\log_b a^n = n\log_b a$,
3. $\log_b a = \dfrac{\log_c a}{\log_c b}$,
4. $\log_b(1/a) = -\log_b a$,
5. $\log_b a = \dfrac{1}{\log_a b}$,
6. $a^{\log_b c} = c^{\log_b a}$,

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \frac{n^2\log(n)}{n\,(\log n)^{10}} = \frac{n}{(\log n)^9} = \text{inf}$$ -> g(n) is **Ω** of f(n)

$$\lim_{N\to\infty}\frac{g(n)}{f(n)} = \frac{n\,(\log n)^{10}}{n^2\log(n)} = \frac{(\log n)^9}{n} = 0$$ -> f(n) is **o** of g(n)

Comparing two functions

Asymptotic Notations are tools for comparing the growth rates of functions.
So sometimes you might need to compare two asymptotic notations.

$f(n) = 3n^{\sqrt{n}}$ $g(n) = 2^{\sqrt{n} \log_2 n}$

$3n^{\sqrt{n}}$ $2^{\log_2 n \sqrt{n}}$ Based on 2 in reverse order.

$3n^{\sqrt{n}}$ $(n^{\sqrt{n}})^{\log_2 2}$ Based on 6

$3n^{\sqrt{n}}$ $n^{\sqrt{n}}$

Ans : f(n) > g(n)

f(n) is Θ (g(n))

1. $\log_c(ab) = \log_c a + \log_c b$,
2. $\log_b a^n = n \log_b a$,
3. $\log_b a = \frac{\log_c a}{\log_c b}$,
4. $\log_b(1/a) = -\log_b a$,
5. $\log_b a = \frac{1}{\log_a b}$,
6. $a^{\log_b c} = c^{\log_b a}$,

Best, Worst and Average Case Analysis

Agenda

- Two examples
  - Linear Search
  - Binary Search Tree
- Best cases & Best case time
- Worst case and Worst case time
  - Minimum time in worst case
  - Maximum time in worst case
- Average case

Example of Best, Worst and Average Case

Linear Search

1	2	2	3	4	5	6	7

- Ex: key = 5, it will find 5 in 6 comparisons
- Ex: key = 20, it will find 20 in 8 (n) comparison and wont find the element
- Best case –
  - When the algorithm will take minimum time
    - Ans: searching key element present at the first index.
  - What time it will take ?
    - Ans: Constant time 1 = O(1)
  - **Best case time – O(1)**
- Worst case
  - When the algorithm will take maximum time
    - Ans: searching key element present at the last index
  - What time it will take
    - Ans: linear – O(n)
  - **Worst case time – O(n)**
- Average Case – All possible case time / no of cases
  - **Average case time for linear search -** 1+2+3 .... +n /n =(n(n+1)/2)/n = (n+1)/2 = **O(n)**
  - **note:** this type of analysis is very difficult and not possible for every cases, we do it rarely, and mostly it will be equivalent to the worst case time
  - Mostly we analyze the **worst case** and **worst case time**.

Don’t be Confused between notation and cases

Cases – Best case ,worst case and average case
Notation – Big O, Omega and Theta $O \quad \Omega \quad \Theta$

Best case
Worst case
Average case

we can write in any notation $O \quad \Omega \quad \Theta$

Both of these concepts are different, so don’t get confused.

Common point of confusion
Worst case doesn't not mean lower bound
Best case doesn’t mean upper bound.

Example of Best, Worst and Average Case

- **Linear Search**

1	2	2	3	4	5	6	7

- Best case –
  - **Best case time – O(1) –** This belong to **constant** class

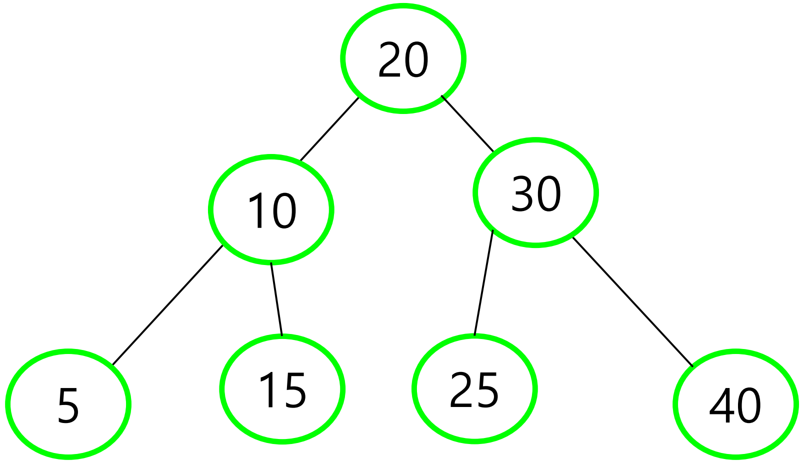
    O (1) **Ω (1)** **Θ (1)**
- Worst case
  - **Worst case time – O(n) –** This belong to **liner** class

    O (n) **Ω (n)** **Θ (n)**

Common point of confusion
Worst case doesn't not mean lower bound
Best case doesn’t mean upper bound.

Example of Best, Worst and Average Case

- Binary Search Tree
  - in binary search tree for every node, all the elements smaller to it will be in the left side and larger to it will be on the right side
  - What is time complexity of searching this tree – 3 . Only 3 comparison is required to search any element



Height of the tree (i.e 3) - log n

If n= 8

$n = 2^3$

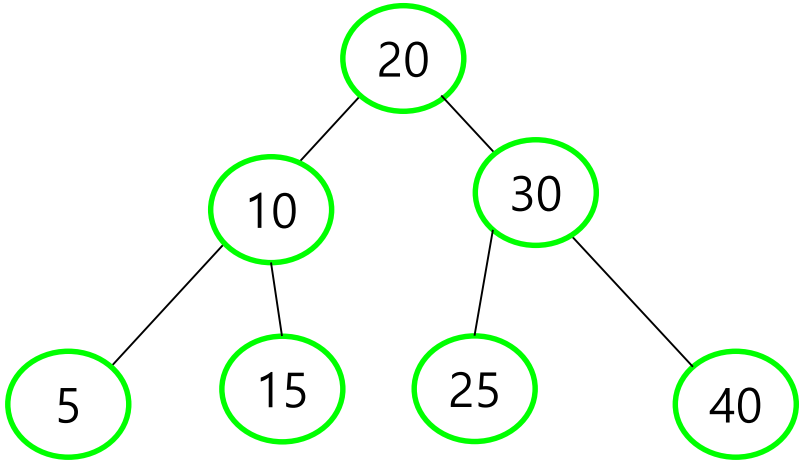
$\log n = \log 2^3$

$\log n = 3$

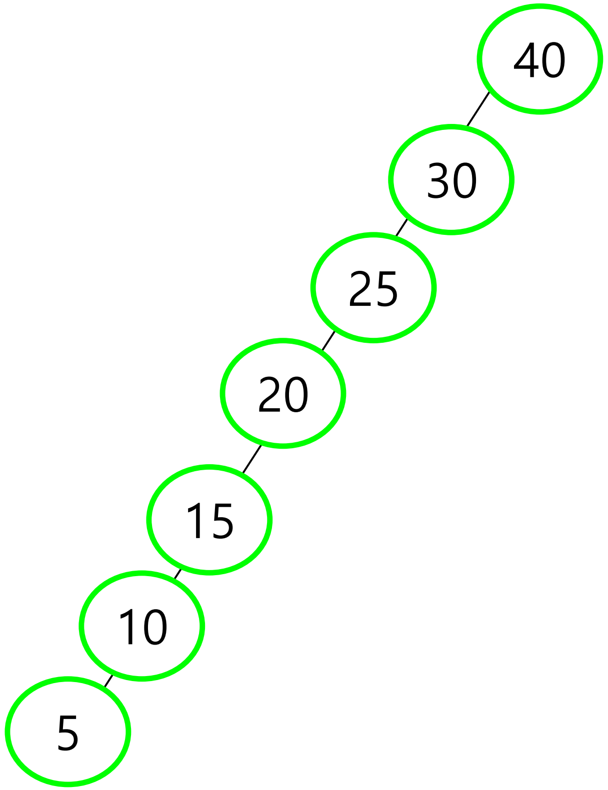
- Best case
  - What will be the best case (best test case for this) ?
    - Searching the element which is at the **root** of this tree, means searching for 20 is best test case
  - How much time it will take to search a root of any binary search tree ?
    - Constant time
  - **Best case - O(1)**
- Worst case
  - What will be the worst case ?
    - Searching the element at the **leaf**
  - How much time it will take to search a leaf of any binary search tree ?
    - log n
  - **Worst case - O(log n)**

Minimum and Maximum Worst case

- Worst Case
  - Minimum worst case
  - Maximum worst case



Worst case is height of the tree - **log n**



Worst case is height of the tree - **n**

- Best case
  - What will be the best case (best test case for this) ?
    - Searching the element which is at the **root** of this tree, means searching for 20 is best test case
  - How much time it will take to search a root of any binary search tree ?
    - Constant time
  - **Best case - O(1)**
- Worst case
  - What will be the worst case ? - Searching the element at the **leaf**
  - How much time it will take to search a leaf of any binary search tree ? Height of the tree
    - Height of these two tree are different
    - Height of a binary search tree could be minimum as log n and maximum as n
  - **Minimum Worst case of Binary search tree - O(log n)**
  - **Maximum worst case of Binary search tree – O (n)**

Note: this is not possible for every data structure and algorithm