Unit 1 : Introduction

Algorithm Efficiency -

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Main problem -

* Bad algo runs faster on small datasets
* Good algo if implemented sloppyily will run slowly
* Scaling problem
  Suppose algo A and B perform comparably on $n=100$
  what happens after $n=1000$.
  A runs faster and B runs slowly

So efficiency definition should be platform independent, instant independent, and have predictive value with respect to increasing input size.

Efficiency (2) - An algo is efficient if it achieves qualitatively better worst-case performance, at an analytical level than brute force search

Efficiency (3) - An algorithm is efficient if it has a polynomial running time.

n^{100} or $n^{1+0.02\log n}$

polynomial   non-polynomial

\# **Worst case Analysis**-

Running time guarantee for any input of size 'n'.

Generally captures efficiency in practice.

Exception:- Some exponential time algorithms are used widely in practice because the worst case instances don't arise.

Ex: simplex algo, k-means algo,

Big-O -

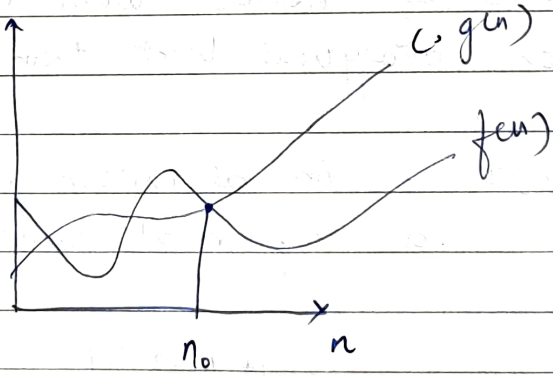
$f(n)$ is $O(g(n))$ exist constant $c > 0$ & $n_0 \geq 0$

such that -

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Ex: $f(n) = 32n^2 + 17n + 1$

$f(n) = O(n^2) \longrightarrow$ select $c = 50$ $n_0 = 1$



Quiz:- $f(n) = 3n^2 + 17n\log n + 1000$

1) $f(n)$ is $O(n^2)$
2) $f(n)$ is $O(n^3)$
3) Both above
4) None.

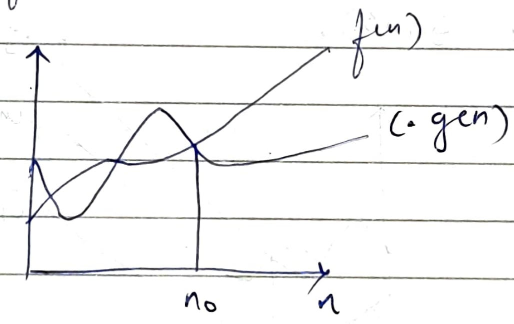
O(g(n)) is a set of function. But we often write f(n) = O(g(n)) instead of f(n) ∈ O(g(n))

Ex : $g_1(n) = 5n^3$ $g_2(n) = 3n^2$
$g_1(n) = O(n^3)$ $g_2(n) = O(n^3)$.
But $g_1(n) \neq g_2(n)$.

This is drawback of Big-O. It is okay to abuse it but not ok to misuse it

Big - Omega
f(n) is $\Omega(g(n))$ if there exist $c > 0$ & $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$,

Ex : $f(n) = 32n^2 + 17n + 1$
$f(n) = \Omega(n^2)$ and $f(n) = \Omega(n)$ $c = 32$
But $n_0 = 1$
$f(n) \neq \Omega(n^3)$



Qu2 : what is an equivalent definition of Omega.
✓ 1) f(n) is Ω g(n) iff g(n) is O f(n).
2) f(n) is Ω g(n) iff c > 0 such that f(n) ≥ c·g(n) for many 'n',
3) Both
4) None.

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

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

If $c_1 = 1/c_2$

$\cancel{g(n) \le}$ $f(n) \ge \frac{1}{c_2} g(n)$.

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

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

$f(n) \ge c_1 g(n)$.

Big Theta

$f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$ $c_2 > 0$ and $n_0 \ge 0$ such that

$c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

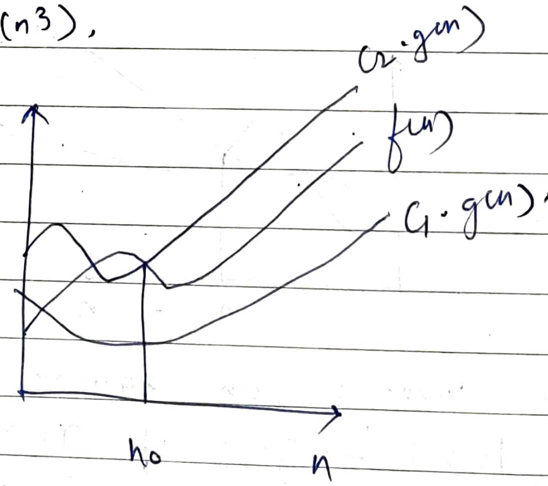
$f(n) = 32n^2 + 17n + 1$

$f(n) = \Theta(n^2)$ — choose $c_1 = 32$, $c_2 = 50$

$n_0 = 1$

$f(n) \ne$ ~~not~~ $\Theta(n)$

$f(n) \ne \Theta(n^3)$.



* Analyzing Algorithm

In algorithm analysis, we try to determine the time taken by algorithm. This time completely depends on the size of the input. Hence, we do all analysis in terms of input size.

The running time is defined as the number of primitive operation or "steps" executed by that algorithm on a particular input. It is important to define the notion of "steps" so that it becomes machine-independent. So we assume that - A constant amount of time is required to execute each line of our pseudocode / algorithm.

Analysis of insertion sort

Insertion sort (A)	cost	times.
for $j = 2$ to A.length	c_1	n
key = A[j]	c_2	$n-1$
// Insert A[j] in its correct position	0	
in A[1 ... j-1]	0	
$i = j-1$	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	
$A[i+1] =$ key	c_8	$n-1$

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

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1).$$

Now, the running time still depends on the ~~type~~ of input values even for fixed input size due to line 5.

<u>for</u> <u>Best case</u> - input is already sorted.

Thus $t_j = 1$ for $j = 2, 3, \ldots n$.

Best case running time is

$$\therefore T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)\, n - (c_2 + c_4 + c_5 + c_8)$$
$$= An + B. \quad (\text{for constant } A \text{ \& } B)$$

↳ A linear function of 'n'.

<u>for</u> <u>worst</u> <u>case</u> - input is sorted in descending order.

Thus $t_j = j$ for $j = 2, 3, \ldots n$.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} j-1 = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left[\frac{n(n+1)}{2} - 1\right]$$
$$+ c_6\left[\frac{n(n-1)}{2}\right] + c_7\left[\frac{n(n-1)}{2}\right] + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} + \frac{-c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

$$= An^2 + Bn + C \quad (\text{for constant } A, B, C)$$

$\hookrightarrow$ A quadratic fuction of 'n'.

for average case

How long does it take to determine where in subarray A [1 – j-1] to insert A [j].

On average, we consider that half elements in A [1 to j-1] are less than A[j] and half elements are greater than A [j]

Thus $t_j = j/2$. Again after solving we get quadratic function of input size.