JCS1403 DESIGN AND ANALYSIS OF ALGORITHMS

10/1/23

UNIT - I INTRODUCTION

Introduction - fundamentals of algorithmic problem solving - important problem types - fundamental data structures - time complexity - space complexity - Analysis Frame work - Asymptotic notation - Basic efficiency classes

Introduction

Need to study algorithm?
- Practical reasons
- Theoretical reasons
- Skill aspect

Practical reasons

Know the standard algorithms, design algorithm and analyse their efficiency.

Theoretical reason

Knowing the problem solving strategies that can be useful regardless of whether the computer is involved or not. algorithmics is called as the study of algorithm which is the core of computer science.

computer programs don't exist without algorithm.

Skill aspect

Helps in developing analytical skills.

Definition for algorithm

It is a sequence of unambiguous instructions for solving a problem (for obtaining a required output for any legitimate input) in a finite amount of time. diagram

$$\begin{tikzcd} & \text{problem} \arrow[d] & \\ & \text{algorithm} \arrow[d] & \\ \text{input} \arrow[r] & \boxed{\text{"computer"}} \arrow[r] & \text{output} \end{tikzcd}$$

Important points

* Non-ambiguity cannot be compromised.
* Range of inputs have to be specified carefully.
* Same algorithm can be represented in different ways.
* several algorithm possible for solving the same problem.

Eg: sorting, searching.

* Algorithm for the same problem can be based on different ideas and can solve the same problem with for different speed.

Eg: Algorithm to find the GCD of two numbers.

gcd (m, n).

gcd (60, 24)

$$\begin{array}{r|l} 2 & 60 \\ \hline 2 & 30 \\ \hline 3 & 15 \\ \hline & 5 \end{array} \qquad \begin{array}{r|l} 2 & 24 \\ \hline 2 & 12 \\ \hline 2 & 6 \\ \hline & 3 \end{array}$$

$2 \times 2 \times 3 \times 5$

$2 \times 2 \times 2 \times 3$

$2 \times 2 \times 3 = 12$.

Three algorithms to solve this problem.

1) Euclid's algorithm
2) consecutive number checking algorithm
3) Middle school procedure.

Euclid's algorithm

Statement:

when $n > 0$,

$$gcd(m, n) = gcd(n, m \bmod n)$$

Example:

$$\begin{aligned} gcd(60, 24) &= gcd(24, 60 \bmod 24) \\ &= gcd(24, 12) \\ &= gcd(12, 24 \bmod 12) \\ &= gcd(12, 0) \\ &= 12 \end{aligned}$$

gcd (46,0) = 46.

Algorithm :

step - 1 : If n = 0, return m as answer and stop otherwise proceed to step 2.

step - 2 : Divide m by n and store the remainder as t

step - 3 : Assign n to m and t to n and proceed to step 1.

pseudocode :

for

```
while n > 0
    t = m mod n
    m = n
    n = t
return m
```

Example :

gcd (70, 28)   m = 70 , n = 28

n > 0, t = 70 mod 28 = 14

m = 28   n = ~~70~~ 14

gcd (28, 14)

m = 28   n = 14

n > 0, t = $\overset{28}{\cancel{14}}$ mod 14 = 0

gcd (14, 0)

n = 0   So gcd (70, 28) = 14.

Consecutive integer checking Algorithm

In Euclid's algorithm if m is less than n the answer won't be available.

To facilitate this condition, the second algorithm was designed.

Algorithm :

Step - 1 : Assign the minimum of m and n to the variable t .

Step - 2 : Divide m by t if the remainder is equal to 0, go to step 3. otherwise go to step 4.

Step - 3 : Divide n by t if the remainder is equal to 0, stop and return t as answer. otherwise go to step 4.

Step - 4 : decrease t by 1 and go to step 2.

Example

gcd (70, 35)

m = 70 , n = 35

t = min (70, 35)

t = 35

m mod t = 70 mod 35

= 0

n mod t = 35 mod 35

= 0

so gcd (70, 35) = 35

gcd (70, 28)

m = 70 , n = 28

t = min (70, 28)

t = 28

m mod t = 70 mod 28

= 14

t = 28 - 1 = 27

t = 27

m mod t = 70 mod 27

= 16

t = t - 1

t = 26

m mod t = 70 mod 26

= 18

t = t - 1

t = 20

m mod t = 70 mod 20

= 10

$t = t - 1$

$t = 14$

$m \bmod t = 70 \bmod 14$

$= 0$

So $\gcd(70, 28) = 14$

Drawback

Algorithm does not work correctly if either m or n is 0.

11/1/23

Middle School procedure

Algorithm

Step 1 : Find the prime factors of m and n

Step 2 : Identify the common factors.

Step 3 : Compute the product of all the common factors and return the product as answer.

Example

Compute $\gcd(60, 24)$

$60 = 2 \times 2 \times 3 \times 5$

$24 = 2 \times 2 \times 2 \times 3$

$$\begin{array}{r|l} 2 & 60 \\ \hline 2 & 30 \\ \hline 3 & 15 \\ \hline & 5 \end{array} \qquad \begin{array}{r|l} 2 & 24 \\ \hline 2 & 12 \\ \hline 2 & 6 \\ \hline & 3 \end{array}$$

common factors $= 2 \times 2 \times 3$

$\gcd(60, 24)$ = product of common factors $= 2 \times 2 \times 3 = 12$

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

21 22 23 24 25

- In iteration - 1, the algorithm eliminates all multiples of the 1st element, 2.

2 3 5 7 9 11 13 15 17 19 21 23 25

- In iteration-2, the algorithm eliminates all multiples of the next element, 3.

2 3 5 7 11 13 17 19 23 25

- In iteration-3, the algorithm eliminates all multiples of the next element, 5.

2 3 5 7 11 13 17 19 23

It is enough to check for the numbers that can be a multiples of less than or equal to square root of n.

Example:

n = 25 check upto multiples of 2, 3 and 5, less than or equal to square root of 25.

19/1/23

Algorithm sieve (n)

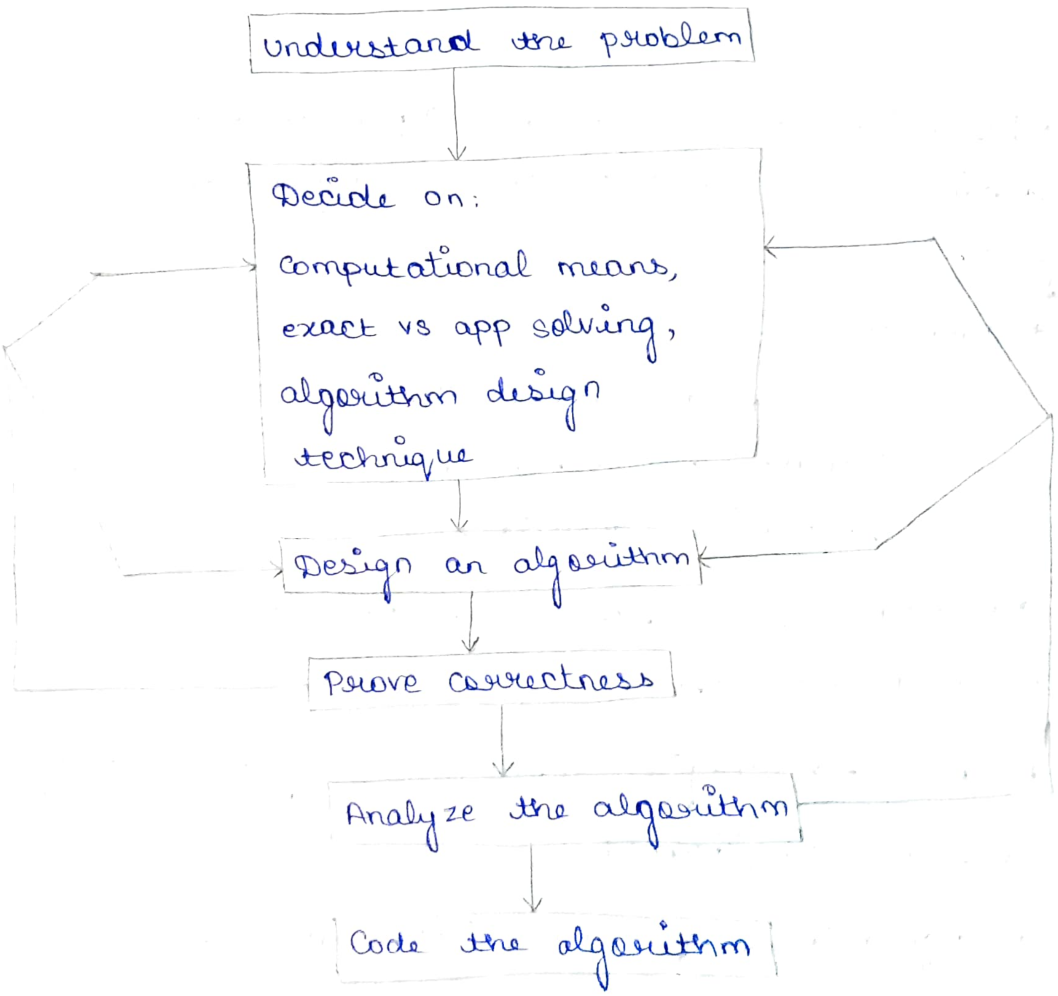
```
// Implements the sieve of Erathosthenes
// Input : A positive integer n>1
// Output : Array L of all prime numbers less than
or equal to n.

for p ← 2 to n do
    A[p] ← p
for  p ← 2 to ⌊√n⌋ do
    if A[p] ≠ 0
        j ← p * p
```

```
while j ≤ n do
        A[j] ← 0
        j ← j + p
```

// Copy the remaining elements of A to array L of the primes.

```
    i ← 0
    for p ← 2 to n do
        if A[p] ≠ 0
            L[j] ← A[p]
            i ← i + 1
    return L
```

Fundamentals of Algorithmic problem solving



A sequence of steps one typically goes through in designing & analyzing an algorithm.

Algorithms → procedural Solutions to problems → Specific instructions for getting answers.

20/1/23
Understanding the problem.

* Read the description
* clarify the doubts by questioning
* If solving a general problem, use known algorithm; otherwise design own, new algorithm.

* Input to an algorithm
* specify the set of instances, otherwise the algorithm may crash on some boundary values.
* correct algorithm

Ascertaining the capabilities of the computational device

* Essential to determine the capabilities of the computational device for which the algorithm is designed.

* computers based on
  von Neumann architecture
  Random-access machine
  sequential algorithms

* Newer computers - concurrent execution of operations. use parallel algorithms.

* speed → today's computers are faster & today's problems are complicated.

* Huge volume of data, critical applications

* Understand the speed & memory available on a Particular computer system.

Choosing between Exact & appropriate problem solving

Exact algorithm → solves the problem exactly

Approximation algorithm → solves the problem approximately - eg. square root, non linear equations, integrals.

Exact solving not possible for many important problems.

Exact algorithms can be unacceptably slow - for complex problems.

21/1/23

Algorithm design techniques

An algorithm design technique → a general approach to solving problems.

Procedural, object oriented, parallel programming, logical programming, functional programming, database oriented

Knowledge of these techniques is important

To provide guidance for designing algorithms for new problems.

To classify algorithms according to a design idea.

Designing an algorithm and data structures

challenging task - choosing proper technique & applying them.

choose appropriate data structures.

Eg. sieve of Eratosthenes.

Methods of specifying an algorithm

2 basic options

words in natural language

Pseudocode

Older technique - flowchart

computer program

Proving an algorithm's correctness

correctness $\rightarrow$ a quality of algorithm $\rightarrow$ algorithm should work correctly for all legitimate inputs.

After specifying an algorithm, prove its correctness $\rightarrow$ sometimes easy & sometimes a complete task.

common technique - mathematical induction - cannot be conclusive.

One instance of its input

Easier for exact algorithm

checked with a predefined limit for approximation algorithm.

Analyzing an algorithm

Qualities of algorithms

- correctness
- Efficiency
    - Time efficiency
    - space efficiency
- simplicity
- generality

coding an algorithm

Has risk and opportunity

Important problem types

- Sorting
- Searching

String processing
graph problems
Combinatorial problems
geometric problems
Numerical problems

sorting

2 properties

stable sorting algorithm   Eg: $8 \; 7 \; \overset{a}{2} \; 12 \; 11 \; 1 \; \overset{b}{2}$

In-place sorting algorithm $1 \; \overset{a}{2} \; \overset{b}{2} \; 7 \; 8 \; 11 \; 12$

searching

sequential & binary search.

string processing

Text strings
Bit strings
Gene sequences
string matching problem

graph problems

shortest-path algorithm
Topological sorting
traveling salesman problem
graph-coloring problem

geometric problems

Applications $\rightarrow$ computer graphics, robotics and tomography.

The closest-pair ~~algorithm~~ problem

The convex-hull problems.

Numerical problems

Approximation problems.
Round-off errors.

24/11/23

Analysis Framework (*) (*)

* Time complexity and space
* Measuring an input's size
* Units of measuring running time
* Orders of growth
* worst-case, Best-Case and Average case efficiencies.
* Recapitulation of the Analysis Framework.

Time complexity.

* Time efficiency, also called the time complexity.
* Today's Computers take less time.
* Measured by counting the no. of times the algorithm's basic operation is executed.

Space complexity

* Space efficiency, also called the space complexity.
* space needed for its i/p & o/p.
* amount of memory units required.
* counting the no. of extra memory units consumed by the algorithm (used for stack during function calls and recursion).

Measuring an input's size

* So use parameter 'n' → input size.
* Algorithms run longer on larger inputs. Eg: Sorting larger arrays etc
* n — problem size (no. of elements in a list, highest degree in a polynomial).
* spell checking algorithm - input size?
  string or text based

* If algorithm examines individual character → No. of characters in the i/p.

* Processes words → No. of words in the i/p

* $b = \lfloor \log_2 n + 1 \rfloor$

Eg: If n = 16 the input size.

$b = \lfloor \log_2 16 + 1 \rfloor = \lfloor \log_2 (2)^4 + 1 \rfloor = 4 + 1 = 5$

Units of Measuring Running time

* Basic operation → contributing the most to the total running time.
* Eg. in sorting → key comparisons
* Eg. in mathematical problems & calculations → arithmetical operations (+, −, *, /).

* Basic operation → The most important operation of the algorithm → the operation contributing the most to the total running time.

* Cop → the execution time of an algorithm's basic operation on a particular computer – assessed approximately.

* C(n) → the number of times this operation needs to be executed for this algorithm – approximate count.

$T(n) \approx C_{op}\, C(n)$

If $C(n) = \frac{1}{2} n(n-1)$ & if n is doubled, how much longer will the algorithm run?

$C(n) = \frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \approx \frac{1}{2} n^2$

if n is doubled

$c(2n) = \frac{1}{2} (2n)^2 = \frac{1}{2} 4n^2$

$$\frac{T(2n)}{T(n)} \approx \frac{C_{op}\, C(2n)}{C_{op}\, C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

25/11/23

Orders of growth

when n increases the function order of growth has an infact and it is considered it assessing the efficiency of algorithm.

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$	$2n$
10		10		100	1000	10^3		20
10^2		10^2		10^4	10^6			2×10^2
10^3		10^3		10^6	10^9			2×10^3
10^4		10^4		10^8	10^{12}			2×10^4
10^5		10^5		10^{10}	10^{15}			2×10^5

* ~~Algor~~ logarithmic function grows slowly.
* 2^n (exponential fn) and n! (factorial fn) grow exponentially.
* when n is doubled, the linear arithmetic function $n\log_2 n$ increases slightly more than double; the quadratic function n^2 & cubic function n^3 increases 4 & 8 times, 2^n & n!
* Algorithms that require an exponential number of operations are practical only for solving problem of very small size. (when n is very small)

worst-case, Best-case & Average-case efficiency

* worst-case efficiency → The efficiency of an algorithm for some i/p of size n for which it takes longer running time.

* worst-case i/p → The i/p for which the algorithm takes the ~~largest~~ longest running time.

* Notation → $C_{worst}(n)$.

* How to determine?

1) Identify the worst case i/p
2) calculate the running time for that worst-case i/p.
3) $C_{worst}(n)$ gives the upperbound for the execution time of any algorithm.

Best-case

* Best-case efficiency → The efficiency of an algorithm ~~for~~ for some i/p of size n, for which it takes the lowest running time.

* Best-case i/p → The i/p for which the algorithm takes the shortest running time.

* Notation → $C_{best}(n)$

* How to determine?

1) Identify the best case i/p
2) Calculate the running time for that best-case i/p.

3) $C_{best}(n)$ gives the lowerbound for the execution time of any algorithm.

Average-case

* Average-case efficiency → The efficiency of an algorithm for any random i/p of size n, for which it takes some running time.

* Average-case i/p → All possible i/p.

* Notation → $C_{avg}(n)$

* How to determine ?

1) Identify all possible i/ps of size n for the algorithm

2) calculate the running time for every possible i/ps and produce the sum as average case efficiency.

3) $C_{avg}(n)$ is between the upper and lower bound of algorithm.

$C_{best}(n) \le C_{avg}(n) \le C_{worst}(n)$

conclusion

To conclude the efficiencies of an algorithm depends on

1) i/p size.
2) specifices of i/ps.

Example :

sequential search

Algorithm sequential search (A[0...n-1], k)

// searches the array A[0...n-1] to check if k is present in the list or not - using linear search.

// Input : Array with n element A[0...n-1] and a search key k.

// output : returns the position of k if it is present in the list, else returns -1 if k is not present in the list.

```
i = 0
while i < n and A[i] ≠ K
    i = i+1
if (i < n)
    return i
else
    return -1
```

Analysis i) worst-case efficiency

when ~~[illegible]~~ k is present in last position of the list (successful search) and when k is not present in the ~~[illegible]~~ list (unsuccessful search). The Algorithm takes n comparisons.

3 5 7 8 1

k = 1 successful search

k = 10 unsuccessful search.

~~so~~ $C_{worst}(n) = n$

Best

when k is present in 1st position of the list the algorithm takes one comparison.

$C_{best}(n) = 1$

27/1/23

Average-case efficiency.

To compute the average-case efficiency for sequential search, the probabilities of successful searches and unsuccessful searches are considered.

The probability of successful search for an element in list of size $n = P/n$.

The probability of unsuccessful search for an element $= 1 - P$

For an element in ith position the algorithm takes i comparison for successful search.

For an element not present in the list - unsuccessful search the algorithm takes n comparison.

$C_{avg}(n)$ = P(successful searches based on comparison) + P(unsuccessful search).

$$= \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \dots + n\frac{p}{n}\right] + n(1-p)$$

$$= \frac{p}{n}(1+2+\dots+n) + n(1-p)$$

$$= \frac{p}{\cancel{n}} \frac{\cancel{n}(n+1)}{2} + n(1-p)$$

$$= \frac{p(n+1)}{2} + n(1-p)$$

For a successful search ~~[illegible]~~ $p=1$ and the alg takes $\frac{n+1}{2}$ comparison. on an average.

For a unsuccessful search $p=0$ and the alg. takes n comparison. on an average.

Basic efficiency classes

Functions can be compared based on orders of growth which may differ by a constant multiple

Example.

* Alg with efficiency n^2, n^3 can be compared.
* Alg with efficiency $\frac{1}{2}n(n-1)$ and n^2 can be compared.

For comparison many different classes are available which can be put under few basic efficiency classes.

The following efficiency classes arranged in increasing order of growth.

Efficiency class	Name	Description
1	Constant	The best case efficiency increases Execution time increases when n increases.
log n	logarithmic	Problem size is cut by a fraction, full i/p is not considered. Eg: binary search.
n	linear	Algorithm that considered a list of size n. Eg: linear search.
n log n	linearithmic	divide and conquer algorithm have this efficiency.
n^2	quadratic	Algorithm with one nested loop. for( ) for( )
n^3	cubic	Algorithm with 2 nested loop. for( ) for( ) for( )
2^n	exponential	Algorithms for which all the subsets of an n element set have to be considered. Eg: Travelling salesman Problem, knapsack Problem.
n!	Factorial	Algorithms that generates all permutation of an n element set. Eg: Job assignment problem.

28/8/23

Asymptotic Notations (*)

Numbers can be compared using symbols like <, ≤, >, ≥, = .

Similarly to compare the efficiency of different alg in terms of their order of growth and to rank them, asymptotic notations can be used.

Three notation used.

Eg.

1. Big oh notation - O $\rightarrow n \in O(n^2)$
2. Big - Omega notation - Ω $\rightarrow n^2 \in \Omega(n)$
3. Big Theta notation - Θ $\rightarrow n^2 \in \Theta(3n^2)$

Big oh notation

O g(n) denotes a set of all functions with lower or same order of growth has g(n).

Eg: $n \in O(n^2)$

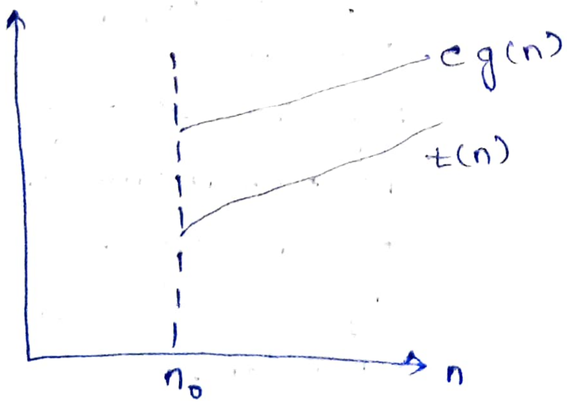
$10n + 8 \in O(n^3)$ $18n^4 + 5n^2 \notin O(n^2)$

$\frac{1}{2}n(n-1) \in O(n^2)$

$\frac{1}{2}n(n-1) \in O(n^3)$

A function t(n) is ~~set~~ said to be in O(g(n)) which can be denoted as $\boxed{t(n) \in O(g(n))}$. if t(n) is bounded above by some constant multiple of g(n) for all large values of n where $n \geq n_0$.

$\boxed{t(n) \leq c\,g(n)}$



Big omega notation

$\Omega g(n)$ denotes a set of all functions with higher or same order of growth has $g(n)$.

Eg: $n^3 \in \Omega(n^2)$

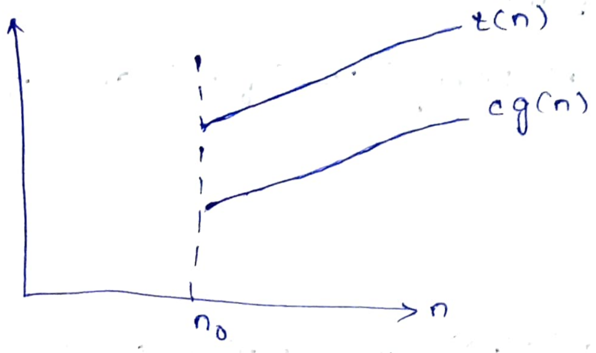
$10n^4 + 8 \in \Omega(n^3)$

$1/2\, n(n-1) \in \Omega(n^2)$

$n^2 \notin \Omega(n^3)$

A function $t(n)$ is said to be in $\Omega(g(n))$ which can be denoted as $\boxed{t(n) \in \Omega(g(n))}$ if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large values of n where $n \geq n_0$.

$$\boxed{t(n) \geq c\,g(n)}$$



31/1/23

Big Theta notation

$\Theta g(n)$ denotes a set of all functions ~~with lower~~ that have ~~and higher or~~ same order of growth ~~has~~ as $g(n)$.

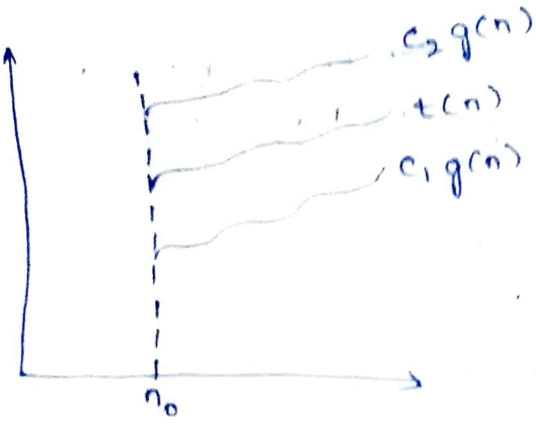
Eg: $n^2 \in \Theta(3n^2)$

$n(n-1) \in \Theta(n^2)$

$3n^3 + 5 \in \Theta(n^3)$

A function $t(n)$ is said to be in $\Theta(g(n))$ which can be denoted as $\boxed{t(n) \in \Theta(g(n))}$. if $t(n)$ is bounded below and above by some constant multiple of $g(n)$ for all large values of n where $n \geq n_0$.

$$\boxed{c_1 g(n) \leq t(n) \leq c_2 g(n)}$$



Properties of Asymptotic notations

1. Reflexivity

$f(n) \in O(f(n))$

Any function $f(n)$ belongs to its own set $O f(n)$.
Since $f(n) \leq c f(n)$.

Similarly $f(n)$ belongs to $\Omega(f(n))$ and $f(n)$ belongs to $\Theta(f(n))$.

2. Symmetry.

$f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$

Eg: $f(n) = n^2$; $g(n) = \frac{1}{2} n(n-1)$

Proof.

$g(n) \in \Theta(f(n))$

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

$\Rightarrow c_1 f(n) \leq g(n), \; g(n) \leq c_2 f(n)$

$\Rightarrow f(n) \leq \frac{1}{c_1} g(n), \; \frac{1}{c_2} g(n) \leq f(n)$

$\Rightarrow \frac{1}{c_1} g(n) \geq f(n) \geq \frac{1}{c_2} g(n)$

If $\frac{1}{c_1} = a_1$ and $\frac{1}{c_2} = a_2$

$\Rightarrow a_2 g(n) \leq f(n) \leq a_1 g(n)$ where a_1 & a_2 are constant.

$\Rightarrow f(n) \in \Theta\, g(n)$

3. Transpose symmetry.

$f(n) \in O\, g(n)$ iff $g(n) \in \Omega(f(n))$

Eg : $f(n) = n^2$, $g(n) = n^3$

$f(n) \in O\, g(n)$

$g(n) \in \Omega\, f(n)$

Proof.

$f(n) \in O\, g(n)$

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

$\frac{1}{c} f(n) \leq g(n)$

$a\, f(n) \leq g(n) \quad (a = \frac{1}{c})$

~~$g(n) \geq$~~

$g(n) \geq a\, f(n)$

$g(n) \in \Omega\, f(n)$.

4. Transitivity

If $f(n) \in O\, g(n)$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

Similarly, if $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$ then $f(n) \in \Omega(h(n))$.

Similarly, if $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$ then $f(n) \in \Theta(h(n))$

Proof.

$f(n) \in O(g(n))$ and $g(n) \in O(h(n))$

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

~~$f(n) \leq a(g(n))$~~ $\qquad$ ~~$g(n) \leq a(h(n))$~~

$\frac{1}{c_1} f(n) \leq g(n)$

$\frac{1}{c_1} f(n) \leq g(n) \leq c_2(h(n))$

$\frac{1}{c_1} f(n) \leq c_2(h(n))$

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

$f(n) \in O(h(n))$

5. If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$

~~space~~

Time complexity

The time taken by implementation of an algorith. is calculated as

$$\underset{\substack{\downarrow\\\text{time}\\\text{taken}}}{t(n)} = \underset{\substack{\downarrow\\\text{compilation}\\\text{time}}}{cp(n)} + \underset{\substack{\downarrow\\\text{Execution}\\\text{time}}}{e(n)}.$$

Space complexity

The overall space used by a program is given as

$$S(p) = \underset{\substack{\downarrow\\\text{fixed}\\\text{component}}}{c(p)} + \underset{\substack{\downarrow\\\text{Variable}\\\text{component}}}{v(p)}$$

$c(p)$ is the space used by the constants, fixed variables and the code.

The variable Component uses space for temporary variables and stack in case of function calls and recursion.