AKSHAYA INSTITUTE OF TECHNOLOGY
Lingapura, Tumkur-Koratagere Road, Tumkur-572106.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Module 1 Notes for

"Design and Analysis of Algorithm" [BCS401]

Prepared by: -
Ms.Trupthi V
Mrs.Ashwini Singh.S
Mrs.Keerthishree PV
Assistant Professors,
Department of CSE.
Akshaya Institute of Technology, Tumakuru

AKSHAYA INSTITUTE OF TECHNOLOGY

Lingapura, Obalapura Post, Koratagere Road, Tumakuru - 572106

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

VISION

To empower the students to be technically competent, innovative and self-motivated with human values and contribute significantly towards betterment of society and to respond swiftly to the challenges of the changing world.

MISSION

M1: To achieve academic excellence by imparting in-depth and competitive knowledge to the students through effective teaching pedagogies and hands on experience on cutting edge technologies.

M2: To collaborate with industry and academia for achieving quality technical education and knowledge transfer through active participation of all the stake holders.

M3: To prepare students to be life-long learners and to upgrade their skills through Centre of Excellence in the thrust areas of Computer Science and Engineering.

Program Specific Outcomes (PSOs)

After Successful Completion of Computer Science and Engineering Program Students will be able to

* Apply fundamental knowledge for professional software development as well as to acquire new skills.
* Implement disciplinary knowledge in problem solving, analyzing and decision-making abilities through different domains like database management, networking, algorithms, and programming as well as research and development.
* Make use of modern computer tools for creating innovative career paths, to become an entrepreneur or desire for higher studies.

Program Educational Objectives (PEOs)

PEO1: Graduates expose strong skills and abilities to work in industries and research organizations.

PEO3: Graduates engage in team work to function as responsible professional with good ethical behavior and leadership skills.

PEO3: Graduates engage in life-long learning and innovations in multi disciplinary areas.

Analysis & Design of Algorithms		Semester	4
Course Code	**BCS401**	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	3:0:0:0	SEE Marks	50
Total Hours of Pedagogy	40	Total Marks	100
Credits	03	Exam Hours	03
Examination type (SEE)	Theory		

Course objectives:

- To learn the methods for analyzing algorithms and evaluating their performance.
- To demonstrate the efficiency of algorithms using asymptotic notations.
- To solve problems using various algorithm design methods, including brute force, greedy, divide and conquer, decrease and conquer, transform and conquer, dynamic programming, backtracking, and branch and bound.
- To learn the concepts of P and NP complexity classes.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) does not mean only the traditional lecture method, but different types of teaching methods may be adopted to achieve the outcomes.
2. Utilize video/animation films to illustrate the functioning of various concepts.
3. Promote collaborative learning (Group Learning) in the class.
4. Pose at least three HOT (Higher Order Thinking) questions in the class to stimulate critical thinking.
5. Incorporate Problem-Based Learning (PBL) to foster students' analytical skills and develop their ability to evaluate, generalize, and analyze information rather than merely recalling it.
6. Introduce topics through multiple representations.
7. Demonstrate various ways to solve the same problem and encourage students to devise their own creative solutions.
8. Discuss the real-world applications of every concept to enhance students' comprehension.

Module-1

INTRODUCTION: What is an Algorithm?, Fundamentals of Algorithmic Problem Solving.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY: Analysis Framework, Asymptotic Notations and Basic Efficiency Classes, Mathematical Analysis of Non recursive Algorithms, Mathematical Analysis of Recursive Algorithms.

BRUTE FORCE APPROACHES: Selection Sort and Bubble Sort, Sequential Search and Brute Force String Matching.

Chapter 1 (Sections 1.1,1.2), Chapter 2(Sections 2.1,2.2,2.3,2.4), Chapter 3(Section 3.1,3.2)

Module-2

BRUTE FORCE APPROACHES (contd..): Exhaustive Search (Travelling Salesman probem and Knapsack Problem).

DECREASE-AND-CONQUER: Insertion Sort, Topological Sorting.

DIVIDE AND CONQUER: Merge Sort, Quick Sort, Binary Tree Traversals, Multiplication of Large Integers and Strassen's Matrix Multiplication.

Chapter 3(Section 3.4), Chapter 4 (Sections 4.1,4.2), Chapter 5 (Section 5.1,5.2,5.3, 5.4)
Module-3
TRANSFORM-AND-CONQUER: Balanced Search Trees, Heaps and Heapsort. **SPACE-TIME TRADEOFFS:** Sorting by Counting: Comparison counting sort, Input Enhancement in String Matching: Horspool's Algorithm. **Chapter 6 (Sections 6.3,6.4), Chapter 7 (Sections 7.1,7.2)**
Module-4
DYNAMIC PROGRAMMING: Three basic examples, The Knapsack Problem and Memory Functions, Warshall's and Floyd's Algorithms. **THE GREEDY METHOD:** Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, Huffman Trees and Codes. **Chapter 8 (Sections 8.1,8.2,8.4), Chapter 9 (Sections 9.1,9.2,9.3,9.4)**
Module-5
LIMITATIONS OF ALGORITHMIC POWER: Decision Trees, P, NP, and NP-Complete Problems. **COPING WITH LIMITATIONS OF ALGORITHMIC POWER**: Backtracking (n-Queens problem, Subset-sum problem), Branch-and-Bound (Knapsack problem), Approximation algorithms for NP-Hard problems (Knapsack problem). **Chapter 11 (Section 11.2, 11.3), Chapter 12 (Sections 12.1,12.2,12.3)**
Course outcome (Course Skill Set) At the end of the course, the student will be able to: 1. Apply asymptotic notational method to analyze the performance of the algorithms in terms of time complexity. 2. Demonstrate divide & conquer approaches and decrease & conquer approaches to solve computational problems. 3. Make use of transform & conquer and dynamic programming design approaches to solve the given real world or complex computational problems. 4. Apply greedy and input enhancement methods to solve graph & string based computational problems. 5. Analyse various classes (P,NP and NP Complete) of problems 6. Illustrate backtracking, branch & bound and approximation methods.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

Continuous Internal Evaluation:

- For the Assignment component of the CIE, there are 25 marks and for the Internal Assessment Test component, there are 25 marks.
- The first test will be administered after 40-50% of the syllabus has been covered, and the second test will be administered after 85-90% of the syllabus has been covered
- Any two assignment methods mentioned in the 22OB2.4, if an assignment is project-based then only one assignment for the course shall be planned. The teacher should not conduct two assignments at the end of the semester if two assignments are planned.
- For the course, CIE marks will be based on a scaled-down sum of two tests and other methods of assessment.

Internal Assessment Test question paper is designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester-End Examination:

Theory SEE will be conducted by the University as per the scheduled timetable, with common question papers for the course (**duration 03 hours).**

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored shall be proportionally **reduced to 50 marks**

Suggested Learning Resources:

Textbooks

1. Introduction to the Design and Analysis of Algorithms, By Anany Levitin, 3rd Edition (Indian), 2017, Pearson.

Reference books

1. Computer Algorithms/C++, Ellis Horowitz, SatrajSahni and Rajasekaran, 2nd Edition, 2014, Universities Press.
2. Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford Stein, 3rd Edition, PHI.
3. Design and Analysis of Algorithms, S. Sridhar, Oxford (Higher Education)

Web links and Video Lectures (e-Resources):

- Design and Analysis of Algorithms: https://nptel.ac.in/courses/106/101/106101060/

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

- Promote real-world problem-solving and competitive problem solving through group discussions to engage students actively in the learning process.
- Encourage students to enhance their problem-solving skills by implementing algorithms and solutions through programming exercises, fostering practical application of theoretical concepts.

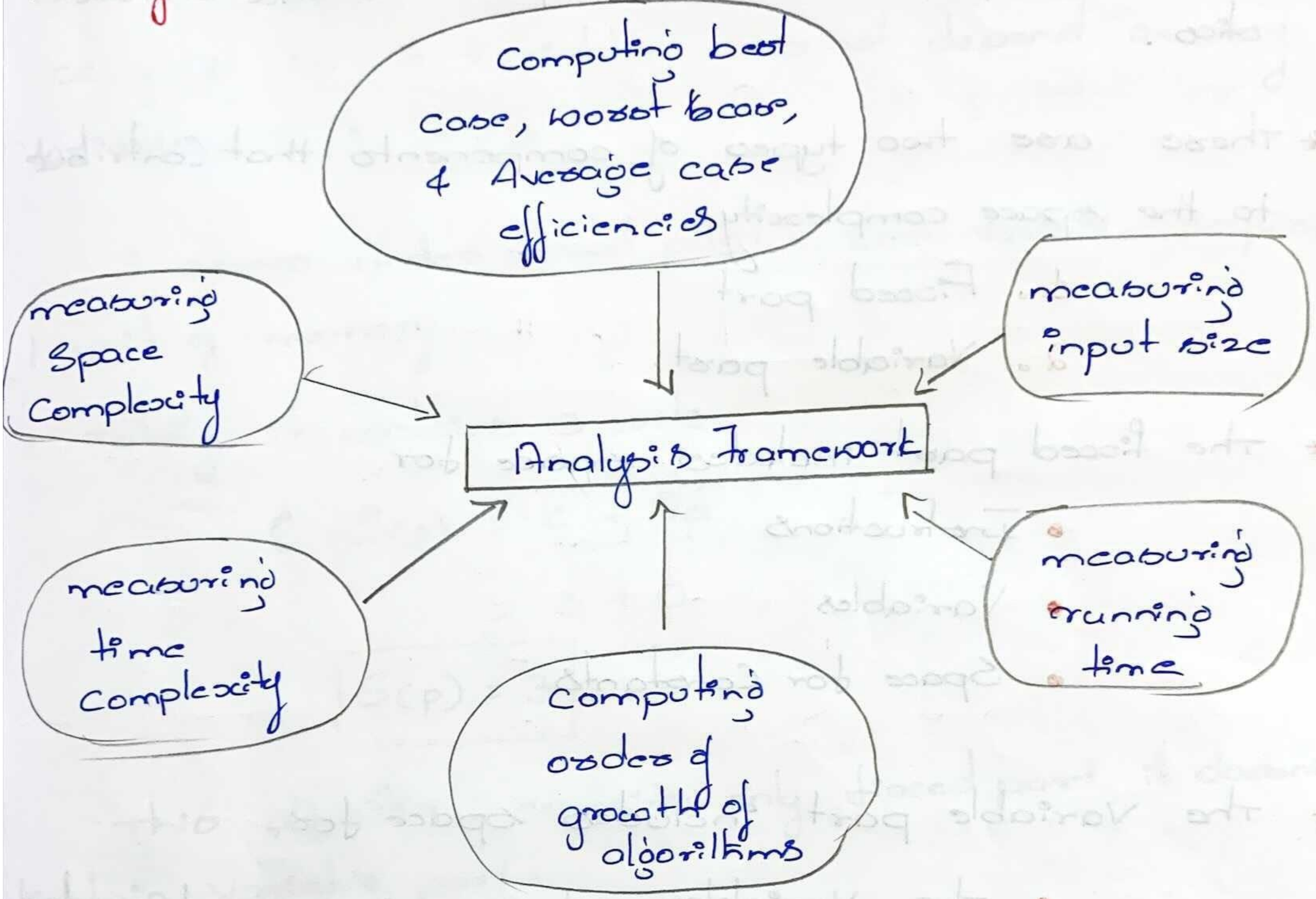
Assessment Methods -

1. Problem Solving Assignments (Hacker Rank/ Hacker Earth / Leadcode)
2. Gate Based Aptitude Test

Module 1

Chapter 2

Analysis Framework



1. Measuring Space Complexity

* the space complexity can be defined as amount of memory required by an algorithm to run
* To compute the space complexity we use 2 factors:
  -> Constant
  -> Instance characteristics.
* the space requirement S(p) can be given as

$$\boxed{S_{(p)} = c + S_p}$$

Where C is a Constant i.e fixed part and it denotes the space of inputs and outputs.

S_p is a space dependent upon instance characteristics.

* There are two types of components that Contribute to the space complexity
  1. Fixed part
  2. Variable part.

* the fixed part includes space for.
  - Instructions
  - Variables
  - Space for Constants

* the Variable part includes space for.
  - the Variables whose size is dependent upon the particular problem instance being solved.

Example:

1. Compute the space required by following algorithm.

Algorithm abc(x y z)
return x*y*z + (x-y).

Sol^n: Algorithm is keyword
abc is name of an algorithm
x, y, z is parameters.

x, y, z is 3 Variables does not depend on any Variables. $\therefore \boxed{C = 3}$

3 comes under fixed part, Each Variable require 1 unit of memory

$\therefore$ Totally it contains 3 units

& $S(p) = c + S_p$

$= 3 + 0$

$\boxed{S(p) = 3}$

This algorithm contains only fixed part. it doesnt Contain Variable part.

2. Algorithm Sum(x, n)

```
{
  total = 0
  for i ← 1 to n do
  {
    total = total + x[i]
  }
}
```

Solⁿ.

Here we have 3 variables x, n, total. $\boxed{C = 3}$

$x[i]$ is Variable part., Array elements depends on Value of 'n'. 'n' units of memory.

$S(p) = C + S_p$

$$\boxed{S_{(p)} = 3 + n}$$

3. Algorithm Recursive

Algorithm Rsum(x, n)
{
if ($n \leq 0$) then
return 0
else
return Rsum ((x, n-1) + x[n])

Solⁿ.

Generally Recursive algorithm uses stack, so, stack requires 3 units of memory

1. Space for formal parameters
2. Space for local variables
3. Space for return address

for each call to Rsum((x, n-1) + x[n])

x requires 1 unit of memory

$n-1$ requires 1 unit of memory

$x[n]$ requires 1 unit of memory

For each call, we require 3 units of me[mory] and we are calling this Rsum, 'n' no of time[s]

Recursive is calling itself, until condition fail
we have to call if statement also

if Statement is calling 1 time $3(n+1)$

∴ this algorithm doesn't contain any fixed

So, $S_{(p)} = C + S_p$

$$\boxed{S_{(p)} = 3(n+1)}$$

Time Complexity

* the time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion

(OR)

* The time complexity can be defined as amount of time required by an algorithm to execute

* It is difficult to compute the time complexity in terms of physically clocked time.

For instance in multiuser system, Executing time depends on many factors, such as -
- → System load
- → Number of other programs running
- → Instruction set used
- → Speed of underlying hardware.

* Time Complexity is therefore given in terms of Frequency Count.

Frequency Count is a count that denotes how many times particular statement is executed.

Example : Consider following code for counting the Frequency Count.

```
Void fun()
{
   int a;
   a = 10; . . . . . . . . . . . . 1
   printf("%d", a) . . . . . . . . 1
}
```

The Frequency count of above program is 2

```
void fun (int a[][] , int b[][])
{
   int c[3][3];
   for (i=0 ; i<m ; i++) . . . . . . . . . m+1
   {
      for (j=0 ; j<n ; j++) . . . . . . . . m(n+1)
      {
         c[i][j] = a[i][j] + b[i][j]; . . . . m·n
      }
   }
}
```

The Frequency count of above program is

$$(m+1) + m(n+1) + mn = 2m + 2mn + 1$$
$$= 2m(1+n) + 1$$

```
Void fun()
{
   int a;
   a=0; ............ 1
   for(i=0; i<n; i++) ....... n+1
   {
      a=a+i; ........ n
   }
   printf("%d", a) ...... 1
}
```

The Frequency count of above code = $\underline{2n+3}$

$1 + n+1 + n + 1$

$2n+3$

```
for(i=1; i<=n; i++) ......... n+1
{
   for(j=1; j<=n; j++) ....... n.(n+1)
   {
      c[i][j]=0; .......... n(n)
      for(k=1; k<=n; k++) ...... n.n(n+1)
         c[i][j]=c[i][j] + a[i][k] * b[k][j];
   }                    ...... n.n.n
}
```

$(2n+1)$ $\quad n(n^2+n)$ $\quad (n^3+n)$

$n+1 + n(n+1) + n.(n) + n.n(n+1) + n.n.n$

$\underline{2n^3 + 3n^2 + 2n + 1}$

Measuring an Input Size

* If the input size is longer, then usually algorithm runs for a longer time.

* Hence we can compute the efficiency of an algorithm as a function to which input size is passed as a parameter.

* To implement an algorithm we require prior knowledge of input size.

  Example, while performing multiplication of 2 matrix we should know order of these matrics. Then only we can enter the elements of matrices.

* Input size is taken as an approximate value. Example. In spell checking algorithms we can predict Exact size of the input

Units for Measuring Running Time

-> the algorithms Execution time can be measured in Seconds, Milliseconds, etc.,

-> this execution time depends on
* Speed of a computer
* Choice of language to implement Algorithm
* Compiler used for generating code.
* Number of inputs.

-> Depending on all such aspects, it is quite difficult to find out the time required then we can't judge which is better one. we have to check the time taken by the algorithm as the input size increase this is known as order of Growth

-> The time which is measured for analyzing an algorithm is generally running time.

-> From an algorithm: we first identify the important operation of an algorithm. this operation is called the basic operation

-> It is not difficult to identify basic operation from an algorithm. Generally the operation which is more time Consuming is a basic operation in the algorithm. Normally such basic operation is located in inner loop

-> For Example, In Sorting algorithm the operation the operation which is Comparing the elements and then placing them at appropriate locations is a basic operation.

-> then we compute total number of time taken by this operation. we can compute the running time of basic operation by given formula.

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

$T(n)$ -> Running time of basic operation

C_{op} -> Time taken by the basic operation to Execute

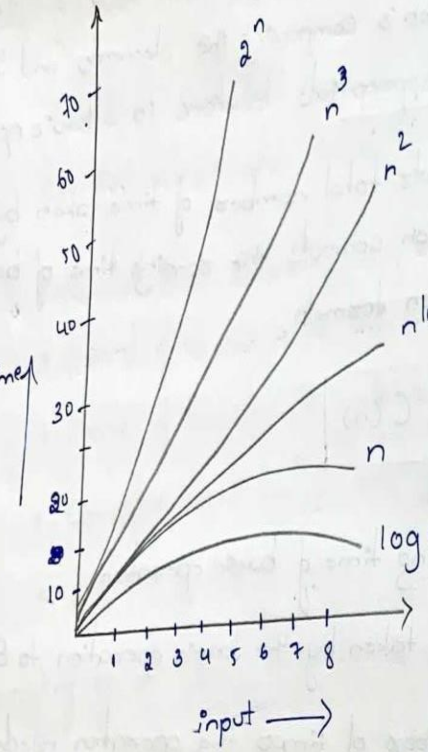
$C(n)$ -> Number of times the operation needs to be Executed.

Order of Growth

Measuring the performance of an algorithm in relation with the input size n is called Order of growth

Ex: the order of growth for varying input size of n is as given below.

n	logn	nlogn	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4294967296



★ From the above graph it is clear that the logarithmic function is the slowest growing function.

★ The exponential function 2^n is fastest and grows rapidly with varying input size n.

★ The Exponential function gives huge values even for small input n.

For the value of $n=16$,

we get $2^{16} = 65536$

Best case, Worst Case & Average case Analysis

Algorithm Seq-search(X[0.... n-1], key)

// Input : An array X [0.... n-1] and search key

// output : Returns the index of X where key value is present

for i←0 to n-1 do

if (X[i]=key) then

return i

Best case time complexity

Best case time complexity is a time Complexity when an algorithm runs for short time.

* In above searching algorithm the element key is searched from the list of n elements.

* If the key element is present at first location in the list (X[0.... n-1]) the algorithm run for a very short time. & then we will get the best case time complexity.

We can denote the Best Case time complexity as

$$\boxed{C_{best} = 1}$$

Worst case time complexity

Worst case time complexity is a time com... when algorithm runs for a longest time.

* In above searching algorithm the element is searched from the list of 'n' elements.

* If the key element is present at n^{th} location then the algorithm will run for longest time. & we will get worst case time complexity.

we can denote the worst case time complexity

$$\boxed{C_{worst} = n}$$

Average case time Complexity

This type of Complexity give information about the behaviour of an algorithm on specific or random input

'p' be a probability of getting successful search

'n' be a is the total number of elements in the list

$\frac{p}{n}$ for every i^{th} location element.

(1-p) be a probability of getting unsuccessful search.

Now, we can find average case time complexity

$$C_{avg}(n) = \text{probability of successful search} + \text{Probability of unsuccessful search}$$

$$C_{avg}(n) = \left[1\cdot\frac{p}{n} + 2\cdot\frac{p}{n} + \ldots\ldots + i\cdot\frac{p}{n}\right] + n\cdot(1-p)$$

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

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

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

Asymptotic Notations

The value of the function may increase or decrease as the value of 'n' increases.

The ~~b~~ asymptotic behavior of a function is the study of how the value of a function varies for large value of n, where n is the size of the input.

Using the asymptotic notation, we can easily find the time efficiency of

Big oh Notation (O)

* the Big oh notation is denoted by 'O'
* It is a method of representing the upper bound of algorithm's running time.

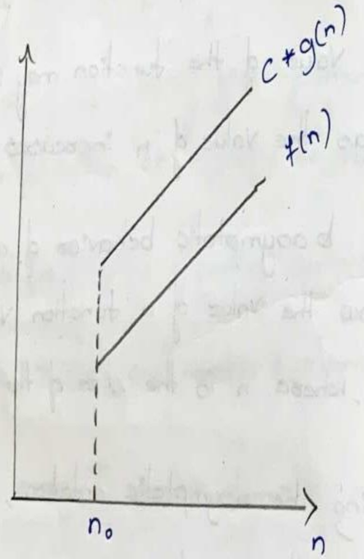
Definition: Let $f(n)$ and $g(n)$ be two non-negative functions. Let n_0 and Constant C are two integers such that n_0 denotes some value of input and $n > n_0$.

ly C is some constant such that $C > 0$.

We can write $\boxed{f(n) \leq C * g(n)}$

* Here $f(n)$ is big oh of $g(n)$. It is also denoted as $f(n) \in O(g(n))$.

Ex:



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

Ex: Consider function $f(n) = 2n+2$ and $g(n) = n^2$. Then we have to find some constant C, so that $f(n) \leq C * g(n)$

Sol^n:

$f(n) = 2n+2 \qquad g(n) = n^2$

For $n=1$, then. $\qquad g(n) = 1^2$

$f(n) = 2(1)+2$

$f(n) = 4$

i.e. $f(n) > g(n)$

$n=2$

$f(n) = 2n+2 \qquad g(n) = n^2$

$f(n) = 2(2)+2 \qquad g(n) = 2^2$

$f(n) = 6 \qquad g(n) = 4$

i.e $f(n) > g(n)$

$n=3$

$f(n) = 2n+2 \qquad g(n) = n^2$

$f(n) = 2(3)+2 \qquad g(n) = 3^2$

$f(n) = 8 \qquad g(n) = 9$

i.e $f(n) < g(n)$ is True

Hence we can conclude that for $n > 2$, we obtain

$\boxed{f(n) < g(n)}$

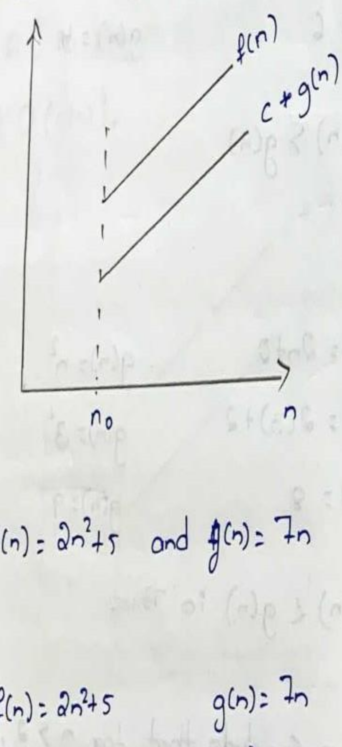
Omega Notation (Ω)

* the omega notation is denoted by 'Ω'
* this notation is used to represent the lower bound of algorithm's running time.

Definition: A function $f(n)$ is said to be in $\Omega(g(n))$ if $f(n)$ is bounded below by some positive Constant multiple of $g(n)$ such that.

$$\boxed{f(n) \geq c * g(n)} \quad \text{for all } n \geq n_0$$

* It is denoted as $f(n) \in \Omega(g(n))$.



Ex:

Consider $f(n) = 2n^2+5$ and $g(n) = 7n$

If $n=0$,

$f(n) = 2n^2+5$ $\quad$ $g(n) = 7n$
$f(n) = 5$ $\quad$ $g(n) = 0$

i.e $f(n) > g(n)$.

But if $n=1$

$f(n) = 2n^2+5$ $\quad$ $g(n) = 7n$
$f(n) = 2(1)^2+5$ $\quad$ $g(n) = 7(1)$
$f(n) = 7$ $\quad$ $g(n) = 7$

i.e., $f(n) = g(n)$

if $n=3$ then,

$f(n) = 2n^2+5$ $\quad$ $g(n) = 7n$
$f(n) = 2(3)^2+5$ $\quad$ $g(n) = 7(3)$
$f(n) = 23$ $\quad$ $g(n) = 21$

i.e $f(n) > g(n)$

Hence we can Conclude that for $n > 3$ we get

$$\boxed{f(n) > c * g(n)}$$

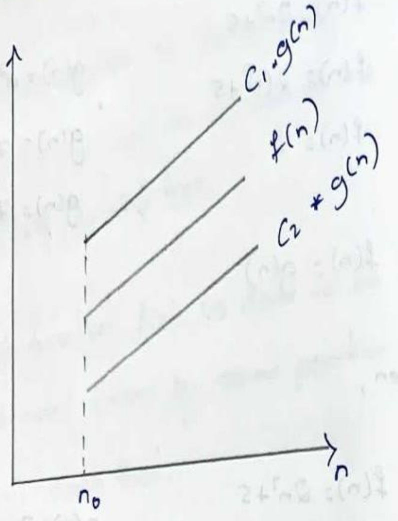
Theta Notation (θ)

* the theta notation is denoted by 'θ'
* By this method the running time is between Upper bound and lower bound.

Definition: Let $f(n)$ and $g(n)$ be two non negative functions. There are two positive Constant namely c_1 and c_2 Such that

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

* It is denoted as $f(n) \in \theta(g(n))$



Ex: If $f(n) = 2n+8$ and $g(n) = 5n$.

where $n \geq 2$

iiiy $f(n) = 2n+8$ and $g(n) = 7n$

Sol^n: $5n < 2n+8 < 7n$ for $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$.

little-oh Notation (o)

* the f is little oh of g as n approaches to n_0 we can written it as

$$f(n) = o(g(n)) \quad \text{when } n \to n_0$$

$$\boxed{\lim_{n \to n_0} \frac{f(n)}{g(n)} = 0}$$

Ex:

1] $n^2 = o(n)$ when $n \to 0$

2] $n \neq o(n^2)$ when $n \to 0$

3] $3n+4 = o(n^2)$ as

$f(n) < c * g(n)$ is always true

But $3n+4 \neq o(n)$.

property of Asymptotic Notation

1. If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then Show that $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$.

Sol^n: By definition we know that, $f(n)$ is said to be big-oh of $g(n)$ and it is denoted by

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

Such that there exists a +ve constant c & n_0

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$

It is given that $f_1(n) \in O(g_1(n))$ there exists a relation $f_1(n) \leq c_1\, g_1(n)$ for $n \geq n_1$ .... (1)

It is given that $f_2(n) \in O(g_2(n))$ there exists a relation $f_2(n) \leq c_2\, g_2(n)$ for $n \geq n_2$ ...... (2)

Let us Assume

$C_3 = \max(C_1, C_2)$ and $n = \max(n_1, n_2)$ ...... (3)

By adding (1) & (2) we have

$$f_1(n) + f_2(n) \le C_1 \cdot g_1(n) + C_2\, g_2(n)$$

$$\le (C_1 + C_2)\,(g_1(n) + g_2(n))$$

$$\le C_3\,(g_1(n) + g_2(n))$$

$$\le C_3 * 2\max(g_1(n), g_2(n))$$

Since $f_1(n) + f_2(n) \le C_3 * 2\max\{g_1(n), g_2(n)\}$

By defination we write

$$\boxed{f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})}$$

2) prove that $f(n) = a_m n^m + \ldots\ldots a_1 n + a_0$, then $f(n) = O(n^m)$

Sol^n Let,

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + a_{m-2} n^{m-2} + \ldots\ldots + a_0 \text{ --- (1)}$$

If we treate a_m^i as a constant then the equation (1) becomes

$$f(n) = A \sum_{i=0}^{m} a^m \text{ --- (2)}$$

Where A is $a_0^0 + a_1^1 + a_2^2 + \ldots\ldots$

If Equation (2) is

$$f(n) = A\left[\sum_{i=0}^{n} 1\right]$$

$$= A[1+1+1 \ldots\ldots +1]$$

$$= A(n^1)$$

$$f(n) = A \cdot O(n^1)$$

If Equation (2) is

$$f(n) = A \sum_{i=0}^{n} n$$

$$= A[1 + 2 + 3 + \ldots\ldots n]$$

$$= A(n^2)$$

$$f(n) = A \cdot O(n^2)$$

If Equation (2) is

$$f(n) = A \sum_{i=0}^{n} n^2$$

$$= A[1^2 + 2^2 + 3^2 + \ldots\ldots n^2]$$

$$= A(n^3)$$

$$f(n) = A \cdot O(n^3)$$

thus

$$f(n) = A \sum_{i=0}^{m} n$$

$$= A\,O(n^m)$$

thus neglecting the constant term we will have

$$f(n) = O(n^m)$$

3] prove $3n^3 + 2n^2 = O(n^3)$; $3^n \neq O(2^n)$

Sol^n: $f(n) \leq c * g(n)$

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

Then,

In short we will find the values of n, c such that

$f(n) \leq c * g(n)$ remains true.

Assume $f(n) = 3n^3 + 2n^2$

$g(n) = n^3$

then for $n >= 2$ and $c = 4$ $f(n) \in O\,g(n)$ is true

i.e $n = 2$ and $c = 4$

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

$= 3(2)^3 + 2(2^2)$ $\quad$ $g(n) = 8$

$f(n) = 32 \longrightarrow$ LHS

$c * g(n) = 4 \times 8$

$= 32$ —— RHS

LHS = RHS is thus proved

But when

$f(n) = 3^n$

$g(n) = 2^n$ then let us find

$3^n \leq c * 2^n$

i.e $\frac{3^n}{2^n} \leq c = \left[\frac{3}{2}\right]^n \leq c$

But there is no such value of c which is $\geq \left(\frac{3}{2}\right)^n$.

Hence $3^n \neq O(2^n)$.

In other words $3^n < c * (2^n)$ will never be true

Order of growth using limits

Big-oh, Big-Omega, and Big-theta can be defined as

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{implies that } f(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & \text{implies that } f(n) \text{ has the same order of growth as } g(n) \\ \infty & \text{implies that } f(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} f(n) \in O(g(n)) \\ f(n) \in \Theta(g(n)) \\ f(n) \in \Omega(g(n)) \end{cases}$$

Formulas :

L-'Hopital rule $\rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

Sterling's rule $\rightarrow n! = \sqrt{2\pi n}\,(n/e)^n$

1. compare order of growth of $\log_2(n)$ and $\sqrt{n}$

Sol^n:

If $n=2$,

$\log_2(n) = \log_2(2) = 1$

$\sqrt{n} = \sqrt{2} = 1.414$

If $n=64$

$\log_2(n) = \log_2(64) = 6$

$\sqrt{n} = \sqrt{64} = 8$

If $n=256$

$\log_2(n) = \log_2(256) = 8$

$\sqrt{n} = \sqrt{256} = 16$

All these Computations Show that.

$$\boxed{\log_2(n) \leq \sqrt{n}}$$

2. Compare order of growth of $\frac{1}{2}n(n-1)$ and n^2.

Sol^n:

If $n=2$

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

$= 1$

$n^2 = (2)^2$

$= 4$

If $n=4$

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

$= 6$

$n^2 = (4)^2$

$= 16$

If $n=8$

$\frac{1}{2}n(n-1) = \frac{1}{2}8(8-1)$

$= 28$

$n^2 = (8)^2$

$= 64$

All Such comparisons indicate that

$$\boxed{\frac{1}{2}n(n-1) < n^2}$$

Basic Efficiency classes

Name of Efficiency class	Order of growth	Description	Example
Constant	1	As input size grows then, we get larger running time	Scanning array elements.
linear	n	the running time of algorithm depends on the input size n.	Sequential Search
nlogn	nlogn	Some instance of input is considered for the list of size n.	sorting the elements using merge sort or quick sort
Quadratic	n^2	When the algorithm has 2 nested loops then this type of efficiency occurs	Scanning matrix multiplication elements
cubic	n^3	When the algorithm has 3 nested loops the this type of Efficiency occurs	performing matrix multiplication
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs	Generating all subsets of n elements.
Factorial	$n!$	When an algorithm is computing all the permutations then this type of efficiency occurs	Generating all permutations
logarithmic	logn	When we get logarithmic running time then it is sure that the algorithm does not consider all its input rather ~~than~~ the problem is divided into smaller parts on each iteration	performing binary search operation

Mathematic Analysis of Non-Recursive Algorithms

General plan for Analyzing the efficiency of non-recursive algorithm. :

1. Decide the input size based on parameter 'n'.
2. Identify Algorithms basic operations.
3. Check how many times the basic operations is Executed. Then find whether the execution of basic operations depends upon the input size n.
4. Setup a sum for the no of times the basic operation is Executed
5. Simplify the sum using standard formula and rules

The Formula used For Analysis are :

1. $\sum_{i=1}^{n} c a_i = c \sum_{i=1}^{n} a_i$

2. $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$

3. $\sum_{i=1} 1$ = upperlimit − lowerlimit + 1

5. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2$

$= \frac{n(n+1)(2n+1)}{6}$

$= \frac{1}{3} n^3$

1. Finding the largest element in a list of 'n' numbers

Algorithm MAX_ELEMENT(A[0.....n-1])

// problem Description: this algorithm is for finding the maximum value element from the array

// Input: An array of real numbers A[0.....n-1]

// output: the value of largest element in A

```
MAX ← A[0]
    for i←1 to n-1 do
        if A[i] > MAX
            MAX ← A[i]
return MAX
```

Number of times operation Executed $= \sum_{i=1}^{n-1} 1$

$= n-1-1+1$

$T(n) = n-1$

$T(n) = O(n)$

2. Multiplication of two m×n matrices 'A' and 'B'

Algorithm matrixMul(A[i,j], B[i,j])

// Input: m×n matrices A & B

// output: Result matrix C = AB

```
for i←0 to n-1 do
    for j←0 to n-1 do
        C[i,j] ← 0
        for k←0 to n-1 do
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
return C
```

The Basic operation depends on 'n'. It is done for each value of k, i & j so, these 3 statements can be written as

$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1}$

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1}$$

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left[(n-1-0+1) \right]$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= n \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

$$= n \sum_{i=0}^{n-1} \left[(n-1-0+1) \right]$$

$$= n \sum_{i=0}^{n-1} n$$

$$= n^2 \sum_{i=0}^{n-1} 1$$

$$= n^2 \left[(n-1-0+1) \right]$$

$$= n^2 (n)$$

$$= n^3$$

$$\boxed{T(n) = O(n^3)}$$

3. Finding whether all the elements in an given array are distinct or not [uniqueness problem]

Algorithm unique A[(0.......n-1)]

// Input : An array A[0......n-1]

// output : True or False

```
for i ← 0 to n-2 do
{
  for j ← i+1 to n-1 do
  {
    if (A[i] = A[j]) then
      return False
  }
}
return True
```

For the Statement can be written as

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1}$$

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1}$$

$$= \sum_{i=0}^{n-2} \left[(n-1) - (i+1) + 1 \right]$$

$$= \sum_{i=0}^{n-2} \left\{ (n-1-i) \right\}$$

$$= \sum_{i=0}^{n-2} n-1 - \sum_{i=0}^{n-2} i$$

Now, taking $(n-1)$ as Common factor, we can write

$$(n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

Formula

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

$$= (n-1)\left[(n-2)-0+1\right] - \frac{(n-2)(n-1)}{2}$$

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

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

$$= \frac{2(n^2-2n+1) - (n^2-3n+2)}{2}$$

$$= \frac{2n^2-4n+2-n^2-3n+2}{2}$$

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

$$\boxed{T(n) = \Theta(n^2)}$$

Mathematic Analysis of Recursive Algorithm

1. Based on input size,

General plan for Analysing the efficiency of Recursive Algorithm.

1. Decide the input size based on the parameter 'n'
2. Identify algorithm's basic operations.
3. Check how many times the basic operation is Executed, then find whether the Execution of basic operation depends upon the input size 'n'.
4. Obtain a Recurrence relation with appropriate Base Condition.
5. Solve the recurrence relation and obtain the order of growth

1. Computing factorial of some number 'n'

the factorial of some number can be obtained performing repeated multiplication

If n=1 then,

Step 1 : n! = 5!

Step 2 : 4! * 5

step 3 : 3! * 4 * 5

step 4 : 2! * 3 * 4 * 5

step 5 : 1! * 2 * 3 * 4 * 5

step 6 : 0! * 1 * 2 * 3 * 4 * 5

Step 7 : 1 * 1 * 2 * 3 * 4 * 5

Algorithm Factorial (n)

// Input : A non-negative integer n

// output : Returns the factorial Value.

```
if n==0
        return 1
else
        return factorial(n-1) * n
```

Mathematical Analysis

The recursive function call can be formulated as

$$F(n) = F(n-1) * n \quad \text{Where } n>0$$

Then the basic operation multiplications is given as $M(n)$. And $M(n)$ is Multiplication Count to Compute factorial (n).

$$M(n) = M(n-1) + 1$$

Now we will solve recurrence using

- Forward Substitution

$$M(1) = M(0) + 1$$

$$M(2) = M(1) + 1$$
$$= 1 + 1 = 2$$

$$M(3) = M(2) + 1$$
$$= 2 + 1$$
$$= 3$$

- Backward Substitution

$$M(n) = M(n-1) + 1$$
$$= [M(n-1-1) + 1] + 1$$
$$= [M(n-2) + 1] + 1$$
$$= [M(n-2) + 2$$
$$= [M(n-1-2) + 1 + 1] + 1$$
$$= [M(n-3) + 1 + 1] + 1$$
$$= M(n-3) + 3$$

$$\boxed{M(n) = M(n-i) + i}$$

Now let us prove correctness of this formula using mathematical induction as follows.

prove $M(n) = n$ by using mathematical induction

Basis : Let $n=0$ then

$$M(n) = 0$$
$$M(0) = 0 = n$$

Induction: if we assume $M(n-1) = n-1$ then

$$M(n) = M(n-1) + 1$$
$$= n - 1 + 1$$
$$= n$$
$$M(n) = n$$