GITAM
(DEEMED TO BE UNIVERSITY)
(Estd. u/s 3 of the UGC Act, 1956)
VISAKHAPATNAM • HYDERABAD • BENGALURU
Accredited by NAAC with A+ Grade

DESIGN AND ANALYSIS OF ALGORITHMS

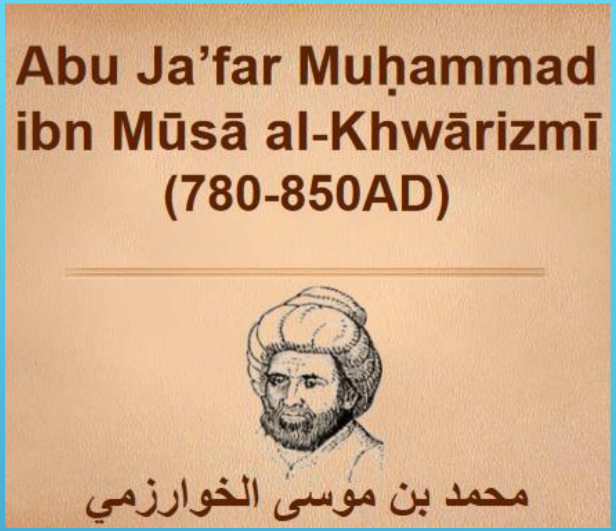
Dr. PV Nageswara Rao
Professor, Dept. of CSE
GIT, GITAM(Deemed to be university)
Visakhapatnam

Lecture 3: Introduction

MODULE - I

- **Introduction:**
  - Algorithm specification
  - Performance analysis
- **Divide and Conquer**
  - The general method, Program Abstraction
  - Binary search
  - Finding maximum and minimum
  - Merge sort
  - Quick sort selection
  - Strassen's matrix multiplication.

ALGORITHM:



- An *algorithm* is a finite set of instructions, that if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:
  - **Input**: Zero or more quantities are externally supplied.
  - **Output**: At least one quantity is produced.
  - **Definiteness**: Each instruction is clear and unambiguous.
  - **Finiteness**: If we trace out the illustration of a an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
  - **Effectiveness**: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper.
- (Algorithms that are *definite* and *effective* are also called as ***computational procedures***. Example: *Operating System*)

STUDY OF ALGORITHMS:

- How to devise algorithms
- How to validate algorithms
- How to analyze algorithms
- How to test a program

- How to devise algorithms:
  - Writing an algorithm is an art which may never be fully automated
  - By mastering some design strategies, new design strategies/algorithms can be devised.
- How to validate algorithms:
  - Algorithm validation is the process of ensuring that it is generating correct results for all possible legal inputs.
  - The algorithm can be verified independently of the programming language
  - Program proving and Program verification is the next step.
    - Program proving or Program verification
      - Annotated by set of assertions about the input and output as expressed in predicate calculus
      - Specification in predicate calclus
- How to analyze algorithms: (Analysis of Algorithms or Performance Analysis)
  - How much computing time and memory/storage required?
  - Quantitative judgement about the value of one algorithm over another.
  - To predict whether the software will meet efficiency constraints that exists.

- How to test a program:
  - **Debugging** is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
  - (Debugging can only points to the presence of errors but not to their absence: E. Dijkstra)
  - A proof of correctness is much more valuable than thousand tests.
  - Two programmers concept.

  - **Profiling**(Performance Measurement) is the process of executing the correct program on data sets and measuring the time and space it takes to compute the results.

ALGORITHM SPECIFICATION:

- Comments, Blocks, Identifiers, data types(not explicitly declared), assignment statement, logical operations, elements of multidimensional arrays and their access, records, sets, lists, looping statements, conditional statements, input and output statements, procedures, arguments passing, etc.

EXAMPLE: SELECTION SORT

```
Algorithm SelectionSort(a,n)
// Sort the array a[1:n] into non decreasing order.
{
  for i := 1 to n-1 do
    {
      lk := i
      for k := i+1 to n do
            If (a[k] < a[lk]  then lk := k;
      t := a[i];   a[i] := a[lk];   a[lk] := t;
    } // end of for loop
  } // end of SelectionSort Algorithm
```

Index	element
1	22
2	27
3	20
4	28
5	12
6	17
7	8
8	23

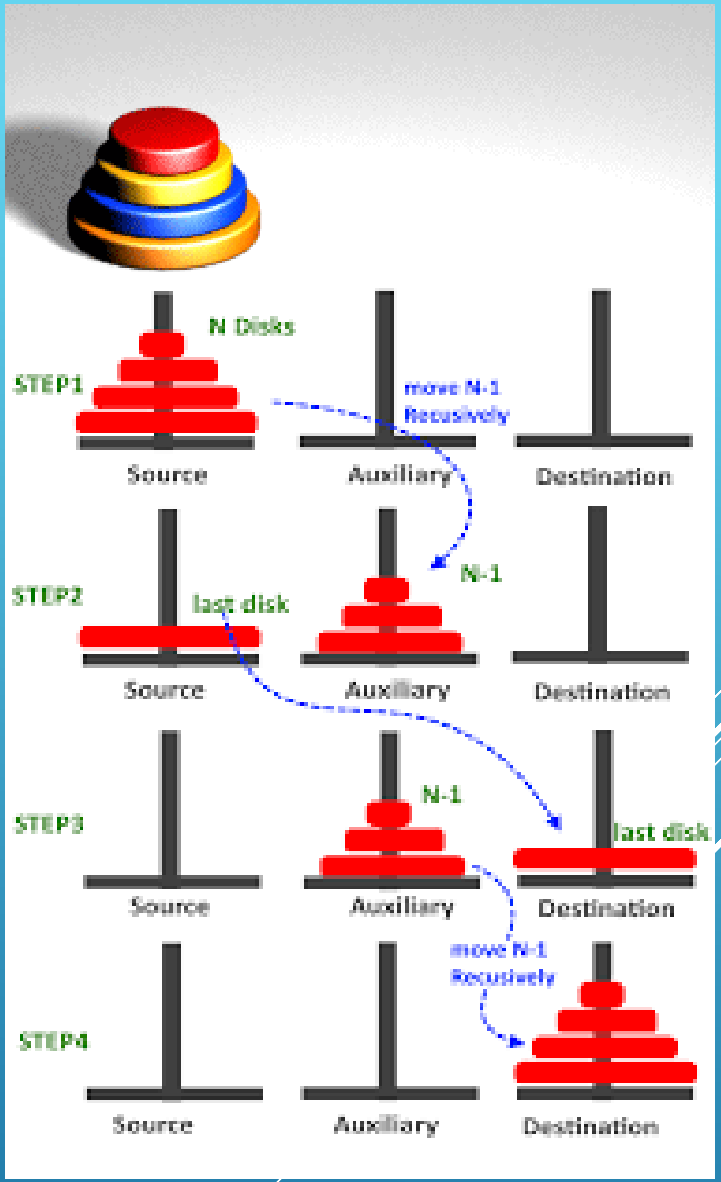
- Comments, Blocks, Identifiers, data types(not explicitly declared), assignment statement, logical operations, elements of multidimensional arrays and their access, looping statements, conditional statements, input and output statements, procedures, arguments passing, etc.

RECURSIVE FUNCTIONS

- A recursive function is a function that is defined in terms of itself.
- Direct recursion
- Indirect recursion / mutual recursion
- Examples: Factorial, Binomial coefficient, Towers of Hanoi, Ackermann's function.

ALGORITHM: TOWERS OF HANOI:

```
Algorithm TowersOfHanoi(n, x, y, z)
// Move the top n disks from tower x to tower y.
{
    if (n ≥ 1) then
    {
        TowersOfHanoi(n − 1, x, z, y);
        write ("move top disk from tower", x,
        "to top of tower", y);
        TowersOfHanoi(n − 1, z, y, x);
    }
}
```



ALGORITHM: ACKERMANN'S FUNCTION

Ackermann's function $A(m, n)$ is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,\ 1) & \text{if } n = 0 \\ A(m-1,\ A(m,\ n-1)) & \text{otherwise} \end{cases}$$

$$\begin{aligned} A(1,2) &= A(0, A(1,1)) \\ &= A(0, A(0, A(1,0))) \\ &= A(0, A(0, A(0,1))) \\ &= A(0, A(0,2)) \\ &= A(0,3) \\ &= 4. \end{aligned}$$

$$\begin{aligned} A(4,3) &= A(3, A(4,2)) \\ &= A(3, A(3, A(4,1))) \\ &= A(3, A(3, A(3, A(4,0)))) \\ &= A(3, A(3, A(3, A(3,1)))) \\ &= A(3, A(3, A(3, A(2, A(3,0))))) \\ &= A(3, A(3, A(3, A(2, A(2,1))))) \\ &= A(3, A(3, A(3, A(2, A(1, A(2,0)))))) \\ &= A(3, A(3, A(3, A(2, A(1, A(1,1)))))) \\ &= A(3, A(3, A(3, A(2, A(1, A(0, A(1,0))))))) \\ &= A(3, A(3, A(3, A(2, A(1, A(0, A(0,1))))))) \\ &= A(3, A(3, A(3, A(2, A(1, A(0,2)))))) \\ &= A(3, A(3, A(3, A(2, A(1,3))))) \\ &= A(3, A(3, A(3, A(2, A(0, A(1,2)))))) \\ &= A(3, A(3, A(3, A(2, A(0, A(0, A(1,1))))))) \\ &= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(1,0)))))))) \\ &= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(0,1)))))))) \\ &= A(3, A(3, A(3, A(2, A(0, A(0, A(0,2))))))) \\ &= A(3, A(3, A(3, A(2, A(0, A(0,3)))))) \\ &= A(3, A(3, A(3, A(2, A(0,4))))) \\ &= A(3, A(3, A(3, A(2,5)))) \\ &= \ldots \\ &= A(3, A(3, A(3,13))) \\ &= \ldots \\ &= A(3, A(3,65533)) \\ &= \ldots \\ &= A(3, 2^{65536} - 3) \\ &= \ldots \\ &= 2^{2^{65536}} - 3. \end{aligned}$$

PERFORMANCE ANALYSIS

- 1. Does it do what we want it to do?
- 2. Does it work correctly according to the original specifications of the task?
- 3. Is there documentation that determines how to use it and how it works?
- 4. Are procedures created in such a way that they perform logical sub-functions?
- 5. Is the code readable?

SPACE/TIME COMPLEXITY

- The ***space complexity*** of an algorithm is the amount of memory it needs to run to completion.
- The ***time complexity*** of an algorithm is the amount of computer time it needs to run to completion.

- Performance Evaluation
  - A priori estimates(performance analysis)
  - A posteriori testing(performance measurement)

SPACE COMPLEXITY $S(P)=C+S_P(I)$

- **Aggregate: Fixed Space Requirements (C)**
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- **Variable Space Requirements ($S_P(I)$)**
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space: formal parameters, local variables, return address

$S(P)=c+S_{P(\text{Instance Characteristics})}$

```
Algorithm abc(a, b, c)
{
    return a + b + b * c + (a + b - c)/(a + b) + 4.0;
}
```

```
Algorithm Sum(a, n)
{
    s := 0.0;
    for i := 1 to n do
        s := s + a[i];
    return s;
}
```

```
Algorithm RSum(a, n)
{
    if (n ≤ 0) then return 0.0;
    else return RSum(a, n - 1) + a[n];
}
```

TIME COMPLEXITY:

- The time T(P) taken by a program P is the sum of the compile time and the run (or execution) time.
- The compile time does not depend on the instance characteristics.
- A compiled program will be run several times without recompilation.
- Consequently we concern ourselves with just the run time of a program.
- This run time is denoted by tp instance characteristics

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

STEP COUNT

- A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

$$\mathbf{return}\ a + b + b * c + (a + b - c)/(a + b) + 4.0;$$

```
Algorithm Sum(a, n)
{
    s := 0.0;
    for i := 1 to n do
        s := s + a[i];
    return s;
}
```

```
Algorithm Sum(a, n)
{
    for i := 1 to n do count := count + 2;
    count := count + 3;
}
```

```
Algorithm Sum(a, n)
{
    s := 0.0;
    count := count + 1; // count is global; it is initially zero.
    for i := 1 to n do
    {
        count := count + 1; // For for
        s := s + a[i]; count := count + 1; // For assignment
    }
    count := count + 1; // For last time of for
    count := count + 1; // For the return
    return s;
}
```

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0;$	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i];$	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n + 3$

s/e : steps per execution

The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.

```
Algorithm RSum(a, n)
{
    if (n ≤ 0) then return 0.0;
    else return RSum(a, n − 1) + a[n];
}
```

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

```
Algorithm RSum(a, n)
{
    count := count + 1; // For the if conditional
    if (n ≤ 0) then
    {
        count := count + 1; // For the return
        return 0.0;
    }
    else
    {
        count := count + 1;   // For the addition, function
                              // invocation and return
        return RSum(a, n − 1) + a[n];
    }
}
```

Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 Algorithm RSum(a,n)	0	–	–	0	0
2 {					
3 if $(n \leq 0)$ then	1	1	1	1	1
4 return 0.0;	1	1	0	1	0
5 else return					
6 RSum$(a, n-1) + a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

```
Algorithm Add(a, b, c, m, n)
{
    for i := 1 to m do
        for j := 1 to n do
            c[i, j] := a[i, j] + b[i, j];
}
```

```
Algorithm Add(a, b, c, m, n)
{
    for i := 1 to m do
    {
        count := count + 2;
        for j := 1 to n do
            count := count + 2;
    }
    count := count + 1;
}
```

```
Algorithm Add(a, b, c, m, n)
{
    for i := 1 to m do
    {
        count := count + 1; // For 'for i'
        for j := 1 to n do
        {
            count := count + 1; // For 'for j'
            c[i, j] := a[i, j] + b[i, j];
            count := count + 1; // For the assignment
        }
        count := count + 1;// For loop initialization and
                           // last time of 'for j'
    }
    count := count + 1;    // For loop initialization and
                           // last time of 'for i'
}
```

Statement	s/e	frequency	total steps
1 Algorithm Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 for $i := 1$ to m do	1	$m+1$	$m+1$
4 for $j := 1$ to n do	1	$m(n+1)$	$mn+m$
5 $c[i,j] := a[i,j] + b[i,j]$;	1	mn	mn
6 }	0	–	0
Total			$2mn+2m+1$

ASYMPTOTIC NOTATION (O)

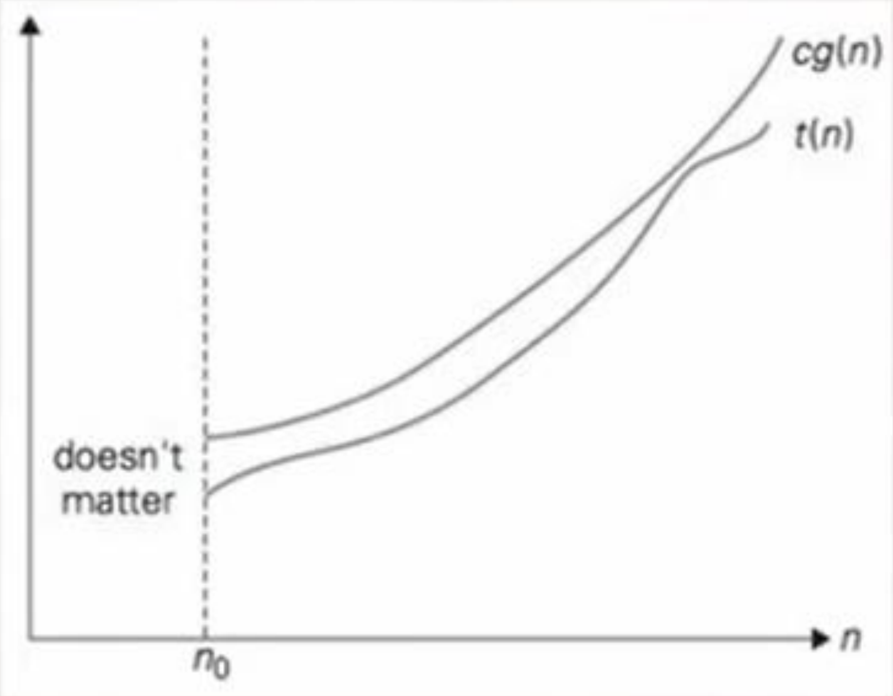
- Definition
- $f(n) = O(g(n))$ (read as: "f of n is big oh of g of n") iff there exist positive constants c and n_0 such that $f(n) \le c * g(n)$ for all n, $n \ge n_0$.
- Examples
  - $3n+2=O(n)$ // $3n+2 \le 4n$ for $n \ge 2$
  - $3n+3=O(n)$ // $3n+3 \le 4n$ for $n \ge 3$
  - $100n+6=O(n)$ // $100n+6 \le 101n$ for $n \ge 6$
  - $10n^2+4n+2=O(n^2)$ // $10n^2+4n+2 \le 11n^2$ for $n \ge 5$
  - $1000n^2+100n-6 = O(n^2)$ // $1000n^2+100n-6 \le 1001n^2$ for $n \ge 100$
  - $6*2^n+n^2=O(2^n)$ // $6*2^n+n^2 \le 7*2^n$ for $n \ge 4$
  - $3n+3=O(n^2)$ // $3n+3 \le 3n^2$ for $n \ge 2$
  - $10n^2+4n+2=O(n^4)$ // $10n^2+4n+2 \le 10n^4$ for $n \ge 2$
  - $3n+2 \neq O(1)$ as 3n+2 is not less than or equal to c for any constant c and all n>n0
  - $10n^2+4n+2 \neq O(n)$
  - Conclusion: The statement f(n)=O(g(n)) states only that g(n) is an upper bound on the value of f(n) for all n, n>=n0.
  - The statement f(n)=O(g(n)) to be informative, g(n) should be as small a function of n as one can come up with f(n)=O(g(n))

Comparing time efficiency

- We measure time efficiency only upto an order of magnitude
  - Ignore constants
- How do we compare functions with respect to orders of magnitude?

Upper bounds, “big O”

- $t(n)$ is said to be $O(g(n))$ if we can find suitable constants c and n_0 so that $cg(n)$ is an upper bound for $t(n)$ for n beyond n_0
  - $t(n) \leq cg(n)$ for every $n \geq n_0$



Examples: Big O

- $100n + 5$ is $O(n^2)$
  - $100n + 5$
  - $\leq 100n + n$, for $n \geq 5$
  - $= 101n \leq 101n^2$, so $n_0 = 5$, $c = 101$
- Alternatively
  - $100n + 5$
  - $\leq 100n + 5n$, for $n \geq 1$
  - $= 105n \leq 105n^2$, so $n_0 = 1$, $c = 105$
- n_0 and c are not unique!
- Of course, by the same argument, $100n+5$ is also $O(n)$

Examples: Big O

- $100n^2 + 20n + 5$ is $O(n^2)$
  - $100n^2 + 20n + 5$
  - $\leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
  - $\leq 125n^2$
  - $n_0 = 1$, $c = 125$
- What matters is the highest term
  - $20n + 5$ dominated by $100n^2$

Examples: Big O

$\exists n_0 \ \forall n \geq n_0 \quad n^3 \leq cn^2$

$n = c$

$c^3 \leq c^3$

$(c+1)^3 > c \cdot (c+1)^2$

- n^3 is not $O(n^2)$
  - No matter what c we choose, cn^2 will be dominated by n^3 for $n \geq c$

Useful properties

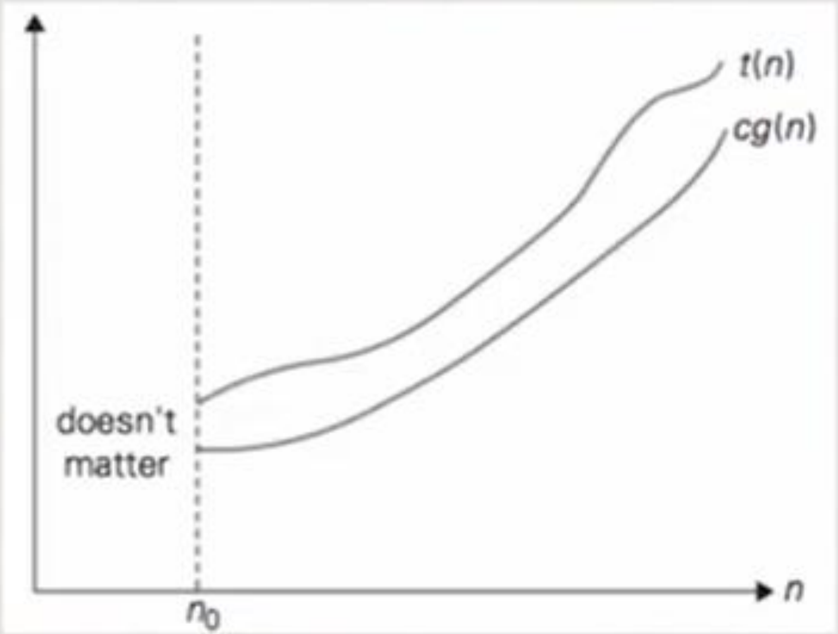
- If
  - $f_1(n)$ is $O(g_1(n))$
  - $f_2(n)$ is $O(g_2(n))$
- then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

Why is this important?

- Algorithm has two phases
  - Phase A takes time $O(g_A(n))$
  - Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time
  - $\max(O(g_A(n)),O(g_B(n)))$
- For an algorithm with many phases, least efficient phase is an upper bound for the whole algorithm

Lower bounds, Ω (omega)

- $t(n)$ is said to be $\Omega(g(n))$ if we can find suitable constants c and n_0 so that $cg(n)$ is an lower bound for $t(n)$ for n beyond n_0
  - $t(n) \geq cg(n)$ for every $n \geq n_0$

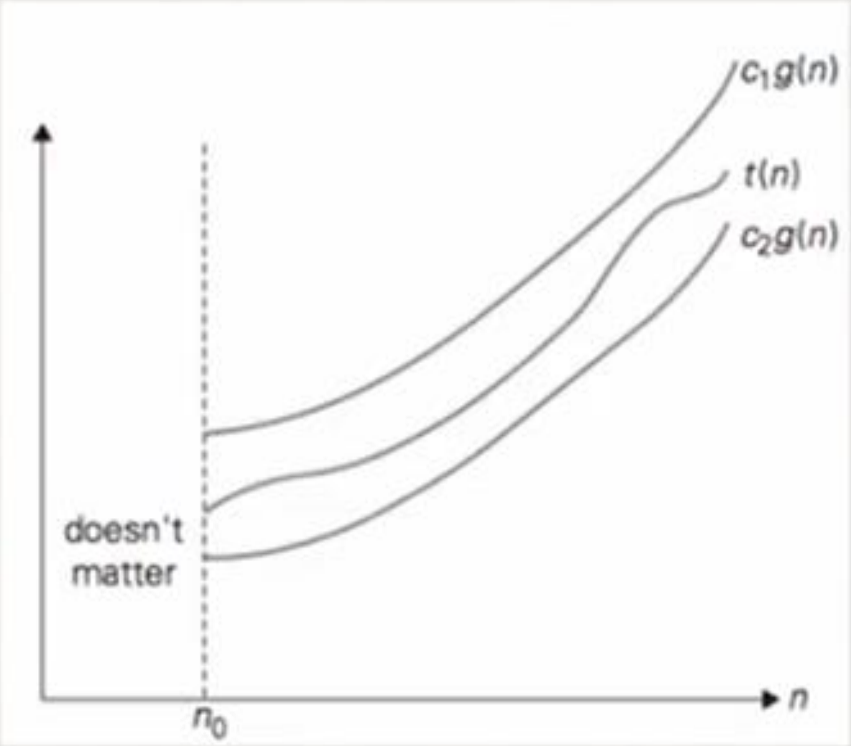


Lower bounds

- n^3 is $\Omega(n^2)$
  - $n^3 \geq n^2$ for all n
  - $n_0 = 0$ and $c = 1$
- Typically we establish lower bounds for problems as a whole, not for individual algorithms
  - Sorting requires $\Omega(n \log n)$ comparisons, no matter how clever the algorithm is

Tight bounds, Θ (theta)

- $t(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$
- Find suitable constants c_1, c_2, and n_0 so that
  - $c_2 g(n) \leq t(n) \leq c_1 g(n)$ for every $n \geq n_0$



Tight bounds

- $n(n-1)/2$ is $\Theta(n^2)$
  - Upper bound

    $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$, for $n \geq 0$
  - Lower bound

    $n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$, for $n \geq 2$
- Choose $n_0 = \max(0,2) = 2$, $c_1 = 1/2$ and $c_2 = 1/4$

Summary

- $f(n) = O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
  - Useful to describe limit of worst case running time for an algorithm
- $f(n) = \Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$
  - Typically used for classes of problems, not individual algorithms
- $f(n) = \Theta(g(n))$: matching upper and lower bounds
  - Best possible algorithm has been found

ASYMPTOTIC NOTATION (Ω)

- Definition
- $f(n) = \Omega(g(n))$ (read as: "f of n is big omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$.
- Examples
  - $3n+2=\Omega(n)$ // $3n+2 \geq 3n$ for $n \geq 1$
  - $3n+3=\Omega(n)$ // $3n+3 \geq 3n$ for $n \geq 1$
  - $100n+6=\Omega(n)$ // $100n+6 \geq 100n$ for $n \geq 1$
  - $10n^2+4n+2=\Omega(n^2)$ // $10n^2+4n+2 \geq 10n^2$ for $n \geq 1$
  - $1000n^2+100n-6 = \Omega(n^2)$ // $1000n^2+100n-6 \geq 1000n^2$ for $n \geq 1$
  - $6*2^n+n^2=\Omega(2^n)$ // $6*2^n+n^2 \geq 2^n$ for $n \geq 1$
  - $3n+3=\Omega(1)$ // even though correct, we never say
  - $10n^2+4n+2=\Omega(1)$ // even though correct, we never say
  - $10n^2+4n+2 = \Omega(n)$ // even though correct, we never say
  - Conclusion: The statement $f(n)=\Omega(g(n))$ states only that g(n) is an lower bound on the value of f(n) for all n.
  - The statement $f(n)=\Omega(g(n))$ to be informative, g(n) should be as large a function of n as one can come up with $f(n)=\Omega(g(n))$

ASYMPTOTIC NOTATION (Θ)

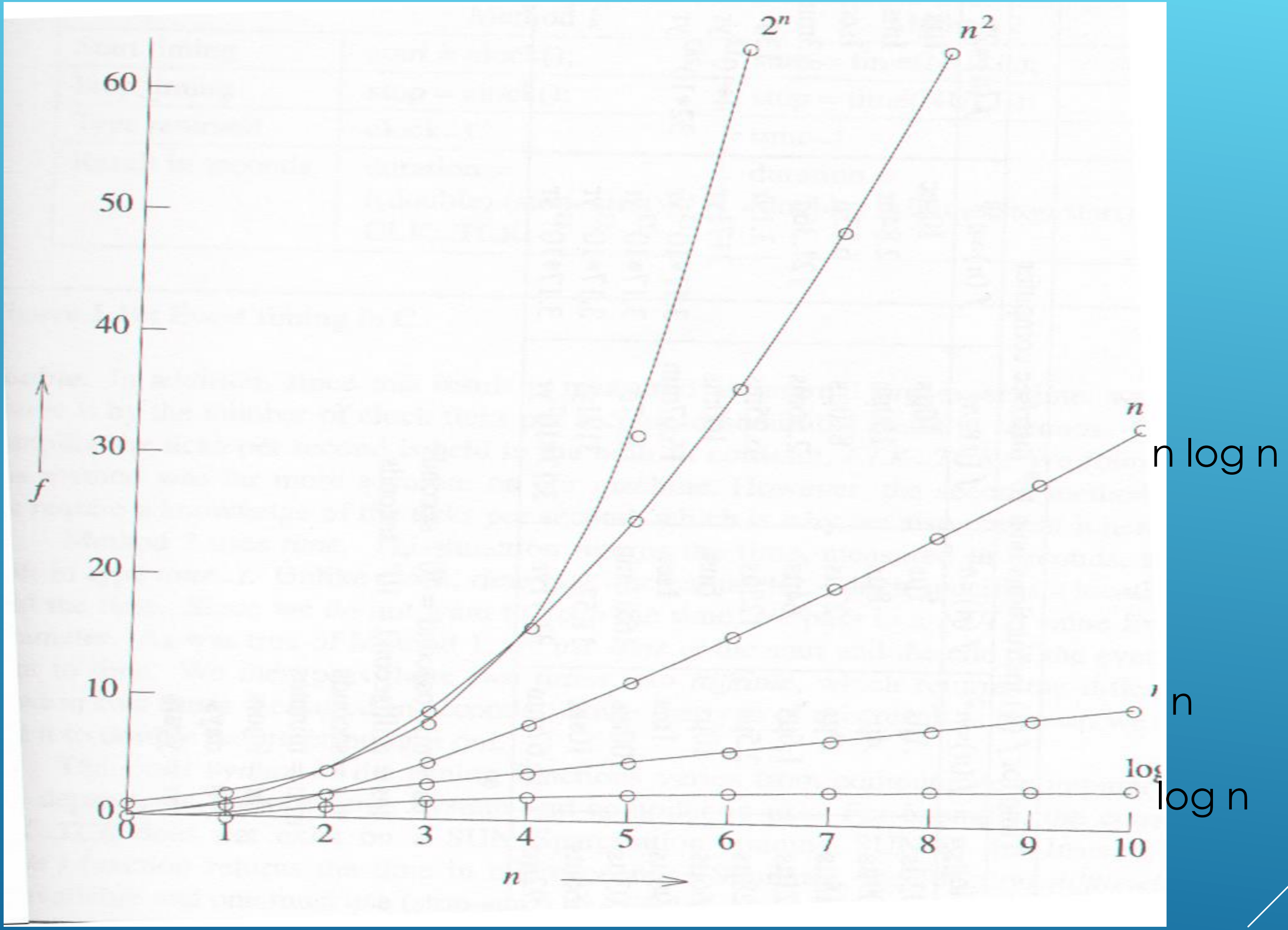
- Definition
- $f(n) = \Theta(g(n))$ (read as: "f of n is big theta of g of n") iff there exist positive constants c1, c2 and n_0 such that $c1*g(n) <= f(n) \le c2*g(n)$ for all n, $n \ge n_0$.
- Examples
  - $3n+2=\Theta(n)$ // $3n+2 \ge 3n$ for all $n \ge 2$ and $3n+2 \le 4n$ for $n \ge 2$ c1=3 , c2=4 and n0=2
  - $3n+3=\Theta(n)$ // 3n+3
  - $10n^2+4n+2=\Theta(n^2)$ $6*2^n+n^2=\Theta(2^n)$
  - $10* \log n +4 = \Theta(\log n)$
  - $3n+2 \ne \Theta(1)$
  - $3n+3 \ne \Theta(n^2)$
  - $10n^2+4n+2 \ne \Theta(n)$
  - The theta notation is more precise than both the big oh and big omega notations.
  - The function $f(n) = \Theta(g(n))$ iff g(n) is both an upper and lower bound on f(n).

- $O(1)$: constant
- $O(\log n)$: Logarithmic
- $O(n)$: linear
- $O(n \log n)$ loglinear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential

FUNCTION VALUES

		Instance characteristic n					
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

PLOT OF FUNCTION VALUES



TIMES ON A 1 BILLION INSTRUCTION PER SECOND COMPUTER

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$ yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$ yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$ yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years

LEARNING OUTCOMES

The student will be able to

- **define** and specify the characteristics of an algorithm. **(L-1)**
- learn **how** to evaluate the performance of an algorithm . **(L-1)**
- **list** different methods in analyzing time complexity. **(L-1)**
- **illustrate** the efficiency of algorithms designed. **(L-2)**